MASTER EN SCIENCES INFORMATIQUES

Preuve de terminaison de boucles
recueil de méthodes et application par l'exemple

de Borchgrave d'Altena, Florence

Award date:
2015

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2014–2015

Preuve de terminaison de boucles : recueil de méthodes et application par l'exemple

Florence de Borchgrave d'Altena

Maître de stage : Fred Mesnard
Co-maître de stage : Etienne Payet

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Wim Vanhoof

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Résumé

Actuellement, il existe une multitude de méthodes qui permettent de prouver la terminaison de boucles. Ce mémoire rassemble, explique et compare, souvent par l'exemple les principales de ces méthodes jugées les plus pertinentes. Notons que ces méthodes peuvent dans certains cas prouver qu'une boucle se terminera. Par contre, aucune d'entre elles n'est capable de prouver la terminaison de toutes les boucles. En effet, nous savons par Turing que la terminaison est un problème indécidable. La plupart des méthodes présentées dans ce mémoire se basent sur la démonstration de Turing prouvant que si une fonction de rang existe, alors la boucle correspondante se terminera. Chacune des méthodes actuelles représente une amélioration d'une méthode précédente. Il n'existe pas encore de méthode universelle.

Mots clés : Terminaison de boucles, méthode de Chen, fonction de rang, preuve de Turing

Abstract

There are currently a high number of methods aiming to demonstrate the termination of a loop. This thesis summarizes, explains, -often by way of examples-, and compares the more important among these methods, those also judged to be the most relevant. Note that these methods may in some cases demonstrate that a loop terminates. However, none of them can prove the termination of every loop. Indeed we know from Turing that issue of loop termination cannot be decided. Most methods presented in this paper are based on the demonstration by Turing that if a ranking function exists, then the corresponding loop will terminate. Each of the current methods represents an improvement over another earlier method. There is however no universal method.

Keywords : Loop termination, Chen's method, ranking function, Turing's proof

Remerciements

Pour réaliser ce mémoire, j'ai effectué un stage de trois mois à l'Université de La Réunion, au sein de la Faculté des Sciences et Technologies et plus précisément dans le laboratoire d'informatique et de mathématique. Ce stage a été supervisé par le Professeur Fred Mesnard, en collaboration avec le Professeur Etienne Payet. Je tiens particulièrement à les remercier pour leur suivi hebdomadaire de mon travail durant tout mon séjour à La Réunion. Leurs précieuses connaissances m'ont permis de me familiariser plus aisément avec les nombreux concepts présents dans mon mémoire. Je tiens également à soulever leurs pertinents conseils lors de l'écriture de la première version de mon mémoire et leur aide concernant l'organisation de mon séjour sur place.

Je tiens aussi à remercier mon promoteur, le Professeur Wim Vanhoof, pour ses riches et nécessaires commentaires lors de l'élaboration et de la rédaction de mon mémoire, ainsi que sa disponibilité.

De plus, je tiens à remercier tous les professeurs et doctorants rencontrés lors de mon parcours universitaire pour les connaissances et conseils qu'ils m'ont apportés d'une façon ou d'une autre. Indirectement, ils ont tous énormément contribué à l'élaboration de ce mémoire.

Je pense aussi à tous les scientifiques et les auteurs qui ont développé, recherché et écrit le contenu des articles qui sont la base de mon mémoire.

Enfin, je remercie également ma famille pour leurs nombreuses relectures et leur patience, ainsi que mes amis pour leur soutien durant toutes mes études et plus particulièrement pendant la réalisation de ce travail.

Table des matières

Introduction

La preuve de terminaison de boucles est décrite et analysée au cours de ce mémoire. Ce sujet est à la fois très précis et très vaste. En effet, il est très vaste, car les boucles se retrouvent dans la plupart des algorithmes informatiques et ces derniers se retrouvent dans presque tous les logiciels. Les logiciels font partie de notre vie quotidienne. Il est donc primordial qu'ils fonctionnent correctement. De plus, il est très précis, car il cible uniquement les boucles.

La terminaison de boucles est une partie de la problématique globale de la vérification de programmes. Actuellement, les programmes informatiques se retrouvent dans énormément de domaines. Prenons par exemple le domaine des transports. À Bruxelles, les métros sont conduits par des chauffeurs, mais par-derrière il y a plusieurs programmes qui permettent aux utilisateurs de connaitre les horaires en temps réel. Quel serait l'impact d'une erreur dans ce programme qui informerait mal l'utilisateur ? Surement rien de catastrophique étant donné que c'est une simple information, cela pourrait cependant irriter l'utilisateur d'avoir été mal informé. À Paris, par contre, les métros roulent sans chauffeur, ils sont entièrement guidés par un programme. Dans ce cas-ci, une erreur dans le programme peut être beaucoup plus catastrophique étant donné que cela pourrait atteindre un bon nombre de vies humaines. C'est dans ce genre de cas que la terminaison de boucles et plus globalement la vérification de programmes prend tout son sens.

Qu'est-ce qu'étudier la terminaison de boucles ? C'est étudier les possibilités de détecter, avant la compilation, si le code source contient une boucle qui ne se termine pas. Autrement dit, il faut s'assurer que les boucles se terminent.

Ce mémoire a pour objectif principal, en rassemblant l'information, de créer un état de l'art de l'existant et de comparer les différentes techniques. De plus, une technique en particulier a été appliquée sur différents exemples et des fonctions ont été développées pour faciliter l'application de cette technique.

Un objectif secondaire de ce mémoire est de rendre plus accessibles ces techniques de terminaison de boucles en les expliquant de manière simple.

Il existe actuellement beaucoup d'articles scientifiques sur ce sujet, mais malheureusement il n'existe que peu d'articles ou autres supports expliquant ces différentes techniques à des novices. Entendons par novice, une personne ayant de bonnes bases en informatique et mathématiques, mais n'ayant jamais approfondi ce sujet en particulier.

Pour répondre aux objectifs, plusieurs méthodes d'application de ces techniques sont expliquées. La plupart d'entre elles sont illustrées par un exemple. Pour commencer, le chapitre 1 expose les différentes notations utilisées tout au long de ce mémoire. Ensuite, un historique des recherches concernant notre domaine d'étude est exposé. Il est suivi d'une catégorisation des méthodes permettant de prouver la terminaison de boucles. Cette catégorisation a pour objectif de familiariser le lecteur pas à pas au principe général de terminaison de boucles appliqué dans ce mémoire. Ces méthodes sont générales et n'ont pas d'algorithme défini pour être implémenté directement.

Suite à la présentation des catégories de méthodes, quatre méthodes actuelles implémentables sont présentées au chapitre 2. Deux de celles-ci utilisent le principe général, par contre deux autres se basent sur des concepts différents. La description de ces deux dernières méthodes a pour objectif de donner une idée générale d'autres techniques possibles n'ayant pas de lien direct avec toutes les autres techniques présentées. De ce fait, elles ne sont que brièvement décrites et nous conseillons au lecteur de se référer à la bibliographie s'il désire approfondir davantage ses connaissances dans ces domaines.

Les deux chapitres suivants (chap. 3 et chap. 4) expliquent en détail, puis appliquent par l'exemple une méthode en particulier. Dans le but de l'appliquer aisément sur différents exemples, cette méthode a été implémentée à l'aide de différents langages. Tous les codes sources développés se trouvent à l'annexe A.

Pour finir, trois récents démonstrateurs sont présentés au chapitre 5. Ces trois démonstrateurs se basent sur le même principe d'une des catégories présentées. L'objectif de ce chapitre est de comparer les implémentations possibles à partir d'un même principe permettant de prouver la terminaison d'une boucle.

Chapitre 1

Catégories de méthodes

Déterminer si un programme se termine est un problème qui existait avant le début de l'informatique moderne. Ce problème de terminaison consiste à déterminer, en un temps fini, si un programme va toujours s'arrêter de s'exécuter ou s'il va potentiellement continuer à s'exécuter à l'infini. Avant toute chose, nous allons définir, à la section 1.1, les conventions et notations utilisées au cours de ce mémoire. Ensuite, avant d'étudier différentes méthodes générales permettant de démontrer la terminaison d'un programme, il est intéressant de faire une brève mise en contexte du problème de terminaison. La section 1.2 présente ce contexte.

Il existe plusieurs méthodes pour prouver la terminaison d'une boucle. Dans les sections suivantes, nous allons présenter trois catégories de méthodes :

1. Méthode intuitive pour prouver la terminaison d'une boucle, à la section 1.3.
2. Méthode traditionnelle pour prouver la terminaison d'une boucle, à la section 1.4.
3. Méthode moderne pour prouver la terminaison d'une boucle, à la section 1.5.

Dans chacune de ces catégories, il existe plusieurs manières de prouver la terminaison d'une boucle. Ces différentes manières représentent chacune une technique précise et elles seront décrites dans la suite de ce mémoire. En effet, à partir du chapitre suivant, des techniques précises permettant de prouver la terminaison seront rigoureusment décrites.

1.1 Conventions et notations

Avant de continuer, il est utile de formaliser les conventions et notations utilisées. Nous utilisons une certaine notation pour les exemples de programmes et les algorithmes définis. Nous la présenterons à la section 1.1.1. De plus, nous utilisons une convention pour nommer les différents types de boucles. En effet, il existe différents types de boucles et il est important de pouvoir les distinguer entre elles car certaines techniques ne peuvent s'appliquer que sur certains types de boucles. Cette convention sera présentée à la section 1.1.2. Enfin, une notation particulière est utilisée pour les variables et les valeurs des variables utilisées dans les programmes et celle-ci sera présentée à la section 1.1.3.

1.1.1 Langage utilisé

Dans ce mémoire, les exemples de programmes et les algorithmes sont présentés en utilisant un pseudo langage fictif et intuitif. Il a été considéré que les lecteurs de ce mémoire auront des connaissances en informatique suffisantes pour qu'il soit inutile, dans le cadre de ce mémoire, de définir formellement ce pseudo langage. Certaines expressions particulières seront expliquées là où elles sont utilisées.

1.1.2 Types de boucles étudiées

Pour étudier la terminaison, il est utile de distinguer différents types de boucles. Ces types seront utilisés dans la suite de ce mémoire. Nous allons distinguer deux types de boucles : les boucles déterministes/non-déterministes et les boucles linéaires simples/complexes.

Boucles déterministes : les boucles dont le corps est composé d'assignations.
Par exemple :

```
int x
while (x > 0)
        x := x − 1
```

Boucles non-déterministes : les boucles dont le corps est composé d'inéquations.
Par exemple :

```
int x
while (x > 0)
        x ≤ x − 1
```

Il peut être étonnant de retrouver une inéquation dans le corps d'une boucle. Pourtant, lors d'une abstraction de programme, le résultat contient souvent ce genre de boucles. Dans ce cas-ci, l'inéquation signifie que lors de la prochaine itération de la boucle, x devra être

décrémenté de minimum une unité. Le x à gauche de l'inéquation représente donc la valeur du x de la prochaine itération, le x à droite représente la valeur du x de l'itération courante.
Par exemple, si $x = 5$, le x de l'itération suivante devra être inférieur ou égal à 4.

Boucles linéaires simples : les boucles dont le corps est composé de simples assignations ou de simples inéquations.
Par exemple :

```
int x
while (x > 0)
        x := x − 2
```

Une boucle linéaire simple, ou en termes anglais « Linear Simple Loop » sera abrégée LSL dans la suite de ce mémoire.

Boucles linéaires complexes : les boucles dont le corps est composé de conditions menant à différentes assignations ou inéquations.
Par exemple :

```
int x
while (x > 0)
        if (x > 10) then x := x − 2
        if (x < 5) then x ≥ x + 1
        otherwise x := x − 5
```

Une boucle linéaire complexe, ou en terme anglais « Linear Complexe Loop » sera abrégée LCL dans la suite de ce mémoire.

1.1.3 Variable et valeur de variable

Dans ce mémoire, les variables et les valeurs des variables utilisées sont présentées en suivant une notation précise.

Les variables sont représentées par des x indexés et leurs valeurs par des a également indexés :

— a^0 représente la valeur d'une variable x avant l'évaluation du test de la boucle.

— a^n représente la valeur d'une variable x après n itérations dans le corps de la boucle.

— Le vecteur A^0 représente toutes les valeurs des variables d'une boucle, avant l'évaluation du test de la boucle.
Formellement, pour un jeu de n variables : $x_1, \ldots, x_n$:

$$A^0 = (a_1^0, \ldots, a_n^0)$$

— Le vecteur A^i représente toutes les valeurs des variables d'une boucle, après i itérations dans le corps de la boucle.
Formellement, pour un jeu de n variables : $x_1, \ldots, x_n$:

$$A^i = (a_1^i, \ldots, a_n^i)$$

— Le vecteur A représente toutes les valeurs des variables d'une boucle, à toutes leurs itérations possibles.
Formellement, pour un jeu de variables : $x_1, \ldots, x_n$ exécutant i itérations :

$$A = (a_1^0, \ldots, a_n^0, a_1^1, \ldots, a_n^1, \ldots, a_1^i, \ldots, a_n^i)$$

Ci-dessous, une illustration des valeurs des variables du programme suivant :

```
int x := 1
int y := 2
while (x < 3)
        x := x + 1
        y := y + 1
```

Analysons les différentes valeurs des variables présentes dans ce programme :

— Valeur de la variable x avant l'évaluation du test de la boucle :

$$a_1^0 = 1$$

— Valeur de la variable y avant l'évaluation du test de la boucle :

$$a_2^0 = 2$$

— Valeur de la variable x après un passage dans le corps de la boucle :

$$a_1^1 = 2$$

— Valeur de la variable y après un passage dans le corps de la boucle :

$$a_2^1 = 3$$

— Valeur de la variable x après deux passages dans le corps de la boucle :

$$a_1^2 = 3$$

— Valeur de la variable y après deux passages dans le corps de la boucle :

$$a_2^2 = 4$$

— Valeur des variables x et y avant l'évaluation du test de la boucle :

$$A^0 = (1, 2)$$

— Valeur des variables x et y après un passage dans le corps de la boucle :

$$A^1 = (2, 3)$$

— Valeur des variables x et y après deux passages dans le corps de la boucle :

$$A^2 = (3, 4)$$

— Valeur des variables x et y durant tous leurs passages dans le corps de la boucle :

$$A = (1, 2, 2, 3, 3, 4)$$

1.2 Contexte de la problématique

Il y a presque un siècle, en 1928, David Hilbert cherchait à mécaniser les mathématiques. Cela l'a amené à se poser la question suivante : « Est-ce que tout est mécanisable ? ». De ce fait, il a commencé à poser le problème de décision (en Allemand « Entscheidungsproblem »). Il utilise un algorithme pour déterminer si une déclaration est universellement vraie. Dans le but de résoudre le problème d'Hilbert ou de montrer qu'il est impossible, les logiciens ont commencé à chercher des instances possibles de problème indécidable.

Turing a prouvé, en 1936, que la terminaison est un problème indécidable (voir [18]). En d'autres termes, il ne peut pas exister de mécanisme/algorithme capable de toujours prouver la terminaison d'un programme. On ne pourrait donc jamais créer une méthode universelle qui prend un quelconque programme en entrée et donne en sortie « se termine » ou « ne se termine pas » en un temps fini. Autrement dit, dans certains cas il est possible de déterminer que le programme se termine ou qu'il ne se terminera pas, mais il est impossible de le faire pour tous les programmes. Heureusement, depuis, on a trouvé différentes techniques qui permettent de prouver la terminaison d'un grand nombre de programmes. Certaines de ces techniques jugées pertinentes sont rassemblées et expliquées dans ce mémoire.

Analysons un exemple qu'il est impossible de traiter, c'est-à-dire une fonction dont on ne sait pas déterminer si elle va se terminer pour toutes ses entrées possibles. Cet exemple provient de la suite de Collatz [8], [13]. Pour construire une suite de Collatz, il faut partir d'un nombre entier strictement positif. Ensuite, s'il est pair il faut le diviser par 2, par contre, dans le cas d'un nombre impair, il faut le multiplier par 3 et ajouter 1. Cette opération est répétée et produit une suite d'entiers positifs dont chacun dépend de son prédécesseur. La conjoncture de Collatz déclare que la suite de Collatz de n'importe quel entier strictement positif, atteint 1. L'équation suivante

définit mathématiquement la suite, pour tout $a_0 \in \mathbb{Z}$:

$$a_n = \begin{cases} \frac{1}{2}a_{n-1} & \text{si } a_{n-1} \text{ est pair} \\ 3a_{n-1}+1 & \text{si } a_{n-1} \text{ est impair} \end{cases}$$

De cette équation, nous pouvons déduire une procédure générant une suite de Collatz pour tout $n \in \mathbb{Z}$ et s'arrêtant lorsque celle-ci atteint 1. Cette procédure reflète donc la conjoncture de Collatz.

```
procedure collatz  n
        if n ≤ 1
                return 1
        else
                if n est pair
                        return collatz n/2
                else
                        return collatz 3n+1
```

À l'heure d'aujourd'hui, la terminaison de ce programme n'a pas encore été prouvée, malgré de nombreuses recherches à ce sujet.

1.3 Méthode intuitive

Cette section présente une méthode intuitive pour prouver la terminaison d'une boucle. Pour des cas extrêmement simples, on peut intuitivement déterminer si le programme se termine. Supposons que nous ayons une boucle très simple où intervient une seule variable dont nous connaissons la valeur. En lisant le code source d'une telle boucle, nous pouvons, par réflexion, connaitre la valeur de la variable lors de ses prochaines itérations dans la boucle. En fonction du test permettant de sortir de la boucle et des valeurs de la variable, il est possible de déduire si la boucle va se terminer.

Pour mieux comprendre et illustrer cette méthode intuitive, nous allons analyser deux exemples de programmes contenant une seule boucle. Le premier se termine, par contre le second ne se termine pas.

1. Exemple d'un programme itératif se terminant :
   x est un entier non borné.

```
int x
read x
if x < 0
        x := -x
while (x > 0)
        x := x - 1
```

   Avant d'entrer dans la boucle, x est positif ou nul. À chaque passage dans la boucle, x est décrémenté. Notons que x est minoré par 0. Après un certain nombre de passages dans la boucle, x vaudra 0. On déduit donc intuitivement qu'on sortira de la boucle.

2. Exemple d'un programme itératif ne se terminant pas :
   k est un entier non borné.

```
int k := 10
while (k > 0)
        k := k + 1
```

   Avant d'entrer dans la boucle, k est positif. À chaque passage dans la boucle, k est incrémenté. Notons que k est minoré par 0. On en déduit intuitivement que la condition de la boucle sera toujours satisfaite et donc que la boucle ne va jamais se terminer.

1.4 Méthode traditionnelle

En 1949, Alan Turing, publie une première preuve de terminaison basée sur une notion d'ordre, appelée les ordinaux. Il publie cette preuve dans l'article « Checking a Large Routine » [19].
Dans un article [12], Robert W. Floyd formalise l'idée de Turing. Il décrit la méthode traditionnelle pour prouver la terminaison d'un programme. Cette méthode se décompose en deux parties :

1. Rechercher un argument de terminaison :
   Le but de cette étape est de rechercher un argument de terminaison sous la forme d'une fonction qui lie chaque état du programme à une valeur dans une structure mathématique. Cette structure mathématique est appelée *un ordre bien fondé*.

2. Tester l'argument de terminaison :
   Si le résultat de la fonction trouvée diminue à chaque transition de programme, l'argument de terminaison sera valide et le programme se terminera. Formellement, soit ρ la fonction trouvée et le programme a une transition de l'état s_1 à l'état s_2, alors il faut que $\rho(s_1) > \rho(s_2)$.
   La fonction ρ est appelée *fonction de rang*.

En d'autres termes, la méthode traditionnelle consiste à trouver une fonction de rang ρ telle qu'à chaque passage dans la boucle la valeur $\rho(x)$ de cette fonction décroît strictement et est minorée par 0. Turing a prouvé dans l'article [19] que si une telle fonction existe, alors la boucle se termine. Cette preuve dépasse le cadre de ce mémoire et nous vous référons donc à son article pour plus de détails. Par contre, s'il est impossible de trouver une telle fonction, alors il est impossible de déduire quoi que ce soit à propos de la terminaison du programme. En effet, dans ce second cas, le programme pourrait tout de même se terminer, donc nous ne pouvons pas affirmer que le programme ne se termine pas.

Dans les chapitres suivants, nous verrons différentes méthodes capables de générer des fonctions de rang. Cette méthode traditionnelle est très importante, car elle est la base de toutes les autres techniques. En effet, les autres techniques précisent comment déduire d'un programme une fonction de rang précise, mais pour déduire la terminaison de ce programme, la plupart font référence à la preuve de Turing.

Pour mieux comprendre et illustrer cette méthode traditionnelle, nous allons analyser deux exemples de programmes. Pour les analyser, il est nécessaire de définir sur quel ensemble bien fondé nous travaillons. Nous allons travailler sur l'ensemble bien fondé : $(\mathbb{N}, >)$. Dans cet ensemble, étant donné qu'une fonction de rang décroît strictement, il faut qu'elle respecte ces propriétés :

$$\begin{cases} \rho(x^0) & \geq & 1 + \rho(x^1) \\ \rho(x^0) & \geq & 0 \end{cases}$$

Rappelons nous, x^0 représente la valeur de x avant d'entrer dans la boucle et x^1 représente la valeur de x à la sortie de boucle.

Voici les deux exemples illustrant cette méthode traditionnelle :

1. Pour ce premier exemple reprenons le programme du premier exemple de la section 1.3.
   Soit le programme suivant, x étant toujours un entier non borné :

```
          int x
          read x
          if x < 0
                    x := −x
(∗)       while (x > 0)
                    x := x − 1
```

   On définit la fonction de rang suivante :

$$\rho(x) = \begin{cases} x & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$$

Cette fonction de rang décroît au fur et à mesure des passages dans la boucle, au point (*), de plus elle est minorée par 0.
Une fonction de rang valide existe, donc la boucle se termine.

2. Pour ce deuxième exemple prenons le programme ci-dessous qui permet de calculer le PGCD de 2 nombres :

```
        int x, y
(*)     while (x > 1, y > 1)
                   if x < y
                              then (x, y) := (x, y − x)
                              else (x, y) := (x − y, y)
```

Cette boucle permet de calculer le PGCD de deux nombres. À chaque passage dans la boucle, au point (*), soit x diminue, soit y diminue. À chaque itération, la somme de x et y va donc décrémenter.
Une fonction de rang possible est :

$$\rho(x, y) = x + y$$

Cette fonction est donc strictement décroissante. Nous avons trouvé une fonction de rang valide, donc la boucle se termine.

1.5 Méthode moderne

Il existe des méthodes plus modernes pour prouver la terminaison d'une boucle. En effet, la méthode traditionnelle pose un problème : trouver une seule fonction de rang pour tout un programme peut s'avérer très difficile. Même pour les programmes les plus simples, cela n'est pas toujours évident. En effet, les fonctions de rang sont parfois utilisées dans des ordres bien fondés plus complexes que les nombres naturels. De ce fait, les recherches de preuves de terminaison ont laissé de côté la recherche d'une seule fonction de rang. À la place, un ensemble de fonctions de rang est cherché. Cela s'appelle un argument de terminaison disjonctif. Les méthodes modernes sont les méthodes se basant sur un tel argument.

— L'avantage des arguments de terminaison disjonctifs est qu'ils sont faciles à trouver, car ils peuvent être exprimés en de petits morceaux indépendants. Chaque morceau peut être trouvé séparément ou incrémentalement en utilisant des méthodes variées et connues pour trouver un unique argument de terminaison.

— Le désavantage des arguments de terminaison disjonctifs est que la condition à vérifier est plus complexe. En effet, il faut considérer toutes les exécutions possibles d'une boucle et non un unique passage dans la boucle. Les récents outils de vérification rendent parfois cette étape possible.

Ce mémoire va présenter et analyser certaines de ces méthodes modernes.

Chapitre 2

Présentation des principales méthodes actuelles

Ce chapitre présente différentes méthodes utilisées actuellement pour déterminer la terminaison d'un programme.

— La section 2.1 présente une méthode complète pour prouver la terminaison de boucles simples publiée par Podelski et Rybalchenko [15].

— La section 2.2 présente une autre méthode pour prouver la terminaison d'une plus large classe de boucles. Cette méthode se base sur des fonctions de rang linéaires à terme [2].

— La section 2.3 présente une toute autre méthode pour prouver la terminaison de boucles. Cette méthode se base sur des fonctions à plusieurs rangs [4].

— La section 2.4 présente deux travaux de Caterina Urban [20] et [21]. Ces travaux se basent sur un domaine abstrait qui permet de démontrer la terminaison de programme par interprétation abstraite.

2.1 Méthode pour prouver la terminaison de la boucle simple

La première méthode présentée consiste à chercher des fonctions de rang linéaires. Elle a été développée par Andreas Podelski et Andrey Rybalchenko [15].

Nous allons tout d'abord décrire la méthode à la section 2.1.1 ensuite nous allons présenter l'algorithme formel de cette méthode à la section 2.1.2. Enfin, nous appliquerons cet algorithme sur un exemple concret de programme à la section 2.1.3.

2.1.1 Description de la méthode

Podelski & Rybalchenko ont développé une méthode complète pour prouver la terminaison de boucles linéaires simples. Cette méthode cherche une fonction de rang linéaire. La méthode est complète donc si une telle fonction existe pour un programme, alors la méthode la trouvera.

Cette méthode fait partie de la catégorie des méthodes traditionnelles pour prouver la terminaison d'une boucle. Nous nous baserons donc sur la preuve de Turing [19] pour prouver la terminaison d'une boucle à partir de la découverte d'une fonction de rang correspondant à cette boucle.

Avant d'étudier cette méthode, il est utile de préciser formellement ce que nous entendons par une fonction de rang linéaire. C'est une fonction linéaire ρ telle que

$$\forall x^0, x^1 : c(x^0, x^1) \Longrightarrow \rho(x^0) \geq 1 + \rho(x^1) \wedge \rho(x^0) \geq 0$$

En d'autres termes, une fonction de rang linaire est une fonction ρ qui part de l'ensemble $\mathbb{Z}$ (resp. $\mathbb{Q}$) pour une boucle dont les variables ont des valeurs dans $\mathbb{Z}$ (resp. $\mathbb{Q}$), vers un ensemble bien fondé. ρ reste positive et décroît strictement d'au moins 1 à chaque itération. Comme l'ensemble d'arrivée de ρ est bien fondé, il représente une fonction de rang.

2.1.2 Algorithme

L'algorithme 2.1 implémente cette méthode. La méthode démontre que : s'il existe des variables λ_1 et λ_2 qui satisfont le système d'inéquations de la ligne de code 9, alors une fonction de rang existe. Rappelons nous qu'il a été démontré (par la méthode traditionnelle) que si une telle fonction existe, alors le programme se termine.

Listing 2.1 Méthode complète pour les fonctions de rang linéaires

1 **input**
2 programme $(A^0 A^1)\begin{pmatrix} X^0 \\ X^1 \end{pmatrix} \leq b$
3 **output**
4 Un booléen :
5 **true** s'il existe une fonction de rang linéaire
6 **false** s'il n'existe pas de fonction de rang linéaire
7 **begin**
8 **if** $\exists \lambda_1, \lambda_2 \in \mathbb{Q}$:
9
$$\begin{cases} \lambda_1, \lambda_2 & \geq & 0 \\ \lambda_1 A^1 & = & 0 \\ (\lambda_1 - \lambda_2) A^0 & = & 0 \\ \lambda_2 (A^0 + A^1) & = & 0 \\ \lambda_2 b & < & 0 \end{cases}$$
10 **then**
11
$$\rho(X^0) = \begin{cases} \lambda_2 A^1 X^0 & \text{s'il existe } X^1 \text{ tel que } (A^0 A^1)\begin{pmatrix} X^0 \\ X^1 \end{pmatrix} \leq b \\ (-\lambda_1 b) - (-\lambda_2 b) & \text{si non} \end{cases}$$
12 **return true**
13 **else**
14 **return false**
15 **end.**

2.1.3 Exemple

L'exemple suivant a pour objectif d'illustrer cette méthode. Reprenons pour exemple le programme suivant :

```
int x
while (x ≥ 0)
        x := x − 1
```

Pour trouver une fonction de rang correspondant à ce programme, nous allons exprimer les relations de transitions de la boucle par un système d'inéquations. Voici le système correspondant à notre programme d'exemple :

$$\begin{cases} x^0 & \geq & 0 \\ x^1 & \geq & x^0 - 1 \\ x^1 & \leq & x^0 - 1 \end{cases}$$

Ce système d'inéquations peut être exprimé sous la forme matricielle :

$$(A^0 A^1) \begin{pmatrix} X^0 \\ X^1 \end{pmatrix} \leq b \tag{2.1}$$

A^0 représente les valeurs des variables de X^0. A^1 celles de X^1. b représente les constantes.

Transformons, tout d'abord, toutes les inéquations sous la forme $a_1 x^0 + a_2 x^1 + \cdots + a_n x^n \leq b_0$. Ceci nous donne :

$$\begin{cases} -x^0 & \leq & 0 \\ x^0 - x^1 & \leq & 1 \\ -x^0 + x^1 & \leq & -1 \end{cases}$$

Ensuite, ces inéquations sont exprimées sous la forme matricielle 2.1 :

$$\begin{pmatrix} -1 & 0 \\ 1 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x^0 \\ x^1 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}$$

Nous avons donc :

$$A^0 = \begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix},\ A^1 = \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix},\ b = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}$$

De plus, car il n'y a qu'une seule variable dans le programme, $X^0 = x^0$ et $X^1 = x^1$.

Dès lors que nous avons calculé les valeurs des variables A^0, A^1 et b, nous pouvons poser $\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3)$ et $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3)$. Ensuite, nous pouvons déduire le système d'inéquations de la ligne de code 9 :

$$\begin{cases} \lambda_1, \lambda_2 \geq 0 \\ -\lambda'_2 + \lambda'_3 = 0 \\ -\lambda'_1 + \lambda''_1 + \lambda'_2 - \lambda''_2 - \lambda'_3 + \lambda''_3 = 0 \\ -\lambda''_1 = 0 \\ \lambda''_2 - \lambda''_3 < 0 \end{cases}$$

Une solution de ce système est :

$$\begin{cases} \lambda_1 & = & (1,0,0) \\ \lambda_2 & = & (0,0,1) \end{cases}$$

Grâce à la ligne de code 11, la fonction de rang de ce programme est :

$$\rho(X^0) = \begin{cases} x^0 & \text{s'il } \exists X^1 \text{ tel que } (A^0 A^1) \begin{pmatrix} X^0 \\ X^1 \end{pmatrix} \leq b \\ -1 & \text{si non} \end{cases}$$

En d'autres termes la fonction de rang est $\rho(x) = x$ et elle va diminuer d'au moins une unité à chaque itération.

En conclusion, cette méthode nous a permis de trouver une fonction de rang linéaire pour notre programme. Par la preuve de Turing [19], nous savons donc que nous pouvons affirmer que dans ce cas, ce programme se termine.

2.2 Méthode visant une plus large classe de boucles

Cette section présente une méthode pour prouver la terminaison d'une plus large classe de boucles, elle se base sur des fonctions de rang linéaires à terme. Elle a été développée par Roberto Bagnara et Fred Mesnard [2].

Nous allons tout d'abord présenter et appliquer, à la section 2.2.2, un algorithme qui prouve la terminaison de boucles LSL au moyen de fonctions de rang linéaires. L'algorithme présenté dans cette section et celui de la section précédente sont équivalents dans le sens où pour une même entrée donnée, ils donnent les mêmes résultats. Cependant, les algorithmes de calcul sont différents.

Dans un deuxième temps, une extension de cette méthode sera présentée et appliquée, à la section 2.2.3. Elle permet de prouver l'existence de fonctions de rang linéaires à terme et donc de prouver la terminaison de plus de boucles.

L'algorithme de cette méthode se base sur le lemme de Farkas. Voici donc en premier lieu, à la section 2.2.1 sa définition.

2.2.1 Lemme de Farkas

Le lemme de Farkas permet d'obtenir des conditions nécessaires et suffisantes pour qu'un système d'équations linéaires ait une solution.

Le lemme de Farkas exprime qu'une inéquation linéaire I (dans les rationnels) est une conséquence logique d'une conjonction satisfiable finie S d'inéquations linéaires, lorsque I est une combinaison linéaire positive de l'inéquation de S.

Formellement :

$$S = \left\{ \begin{array}{ccccccccc} a_{1,1}x_1 & + & \ldots & + & a_{1,n}x_n & + & b_1 & \geq & 0 \\ \ldots & + & \ldots & + & \ldots & + & \ldots & \geq & 0 \\ a_{m,1}x_1 & + & \ldots & + & a_{m,n}x_n & + & b_m & \geq & 0 \end{array} \right.$$

On suppose que S admet au moins une solution.

Soient $x_1, \ldots, x_n$ appartenant au système S, alors le lemme de Farkas établit l'équivalence suivante :

$$(c_1x_1 + \cdots + c_nx_n + d \geq 0) \Leftrightarrow \exists \lambda_1 \geq 0, \ldots, \lambda_m \geq 0. \begin{cases} c_1 &= \sum_{i=1}^{m} \lambda_i a_{i,1} \\ &\vdots \\ c_n &= \sum_{i=1}^{m} \lambda_i a_{i,n} \\ d &\geq \sum_{i=1}^{m} \lambda_i b_i \end{cases}$$

2.2.2 Méthode

Nous allons chercher une fonction de rang linéaire pour une boucle LSL. Ceci est un problème décidable et de complexité polynomiale. Présentons, tout d'abord, la méthode de manière formelle par l'algorithme 2.2.

Listing 2.2 Méthode pour trouver une fonction de rang linéaire

```
1  input
2      Une boucle LSL
3  output
4      Un booléen :
5          true s'il existe une fonction de rang linéaire
6          false s'il n'existe pas de fonction de rang linéaire
7  begin
8      Soit ρ(x,y), une fonction de rang de la forme ax + by.
9      DEC(a) := résultat de l'application du lemme de Farkas pour la partie négative de ρ
10     POS(a) := résultat de l'application du lemme de Farkas pour la partie positive de ρ
11     if DEC(a) ∧ POS(a) est satisfiable
12         return true
13     else
14         return false
15 end
```

Pour illustrer cette méthode, nous allons prendre pour exemple la boucle LSL non déterministe suivante :

```
int x, y
while (x ≥ 0)
        y ≤ −1
        x ≤ x + y
        y ≤ y − 1
```

Commençons par poser le système d'inéquations représentant la boucle. Notons que nous cherchons une fonction de rang de la forme : $\rho(x, y) = ax + by$. Notre objectif est donc de trouver les valeurs de a et de b.
Voici le système d'inéquations représentant la boucle :

$$\begin{cases} x^0 &\geq 0 \\ y^0 &\leq -1 \\ x^1 &\leq x^0 + y^0 \\ y^1 &\leq y^0 - 1 \end{cases}$$

Pour qu'une fonction de rang linéaire existe, il faut que ce système satisfasse les inéquations suivantes :

$$\begin{cases} ax^0 + by^0 & \geq & 1 + ax^1 + by^1 \\ ax^0 + by^0 & \geq & 0 \end{cases}$$

Pour déduire une fonction de rang satisfaisant le système d'inéquations ci-dessus, nous allons appliquer le lemme de Farkas. Pour faciliter les recherches, nous allons travailler séparément sur la partie décroissante et sur la partie positive.

1. Partie décroissante de la fonction de rang :

$$\exists a, b\ \forall x^0, y^0, x^1, y^1 : \begin{cases} x^0 & \geq & 0 \\ y^0 & \leq & -1 \\ x^1 & \leq & x^0 + y^0 \\ y^1 & \leq & y^0 - 1 \end{cases} \Rightarrow ax^0 + by^0 \geq 1 + ax^1 + by^1$$

2. Partie positive de la fonction de rang :

$$\exists a, b\ \forall x^0, y^0, x^1, y^1 : \begin{cases} x^0 & \geq & 0 \\ y^0 & \leq & -1 \\ x^1 & \leq & x^0 + y^0 \\ y^1 & \leq & y^0 - 1 \end{cases} \Rightarrow ax^0 + by^0 \geq 0$$

Maintenant, nous pouvons préparer l'application du lemme de Farkas. Pour cela, nous allons déduire de chaque partie de la fonction de rang quatre nombres rationnels non négatifs pour la partie décroissante (resp. positive) : $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ (resp. $\lambda'_1, \lambda'_2, \lambda'_3, \lambda'_4$).

1. Préparation de l'application du lemme de Farkas pour la partie décroissante :

$$\begin{array}{lcccccccc} \lambda_1 & : & 1x^0 & +0y^0 & +0x^1 & +0y^1 & +0 & \geq & 0 \\ \lambda_2 & : & 1x^0 & +1y^0 & -1x^1 & +0y^1 & +0 & \geq & 0 \\ \lambda_3 & : & 0x^0 & +1y^0 & +0x^1 & -1y^1 & -1 & \geq & 0 \\ \lambda_4 & : & 0x^0 & -1y^0 & +0x^1 & +0y^1 & -1 & \geq & 0 \\ \Rightarrow & & & & & & & & \\ & & ax^0 & +by^0 & -ax^1 & -by^1 & -1 & \geq & 0 \end{array}$$

2. Préparation de l'application du lemme de Farkas pour la partie positive :

$$\begin{array}{lcccccccc} \lambda'_1 & : & 1x^0 & +0y^0 & +0x^1 & +0y^1 & +0 & \geq & 0 \\ \lambda'_2 & : & 1x^0 & +1y^0 & -1x^1 & +0y^1 & +0 & \geq & 0 \\ \lambda'_3 & : & 0x^0 & +1y^0 & +0x^1 & -1y^1 & -1 & \geq & 0 \\ \lambda'_4 & : & 0x^0 & -1y^0 & +0x^1 & +0y^1 & -1 & \geq & 0 \\ \Rightarrow & & & & & & & & \\ & & ax^0 & +by^0 & +0x^1 & +0y^1 & +0 & \geq & 0 \end{array}$$

Dès lors que nous avons ces quatre nombres rationnels non négatifs, nous pouvons appliquer le lemme de Farkas sur chacune des parties.

1. Application du lemme de Farkas pour la partie décroissante :

$$\exists \lambda_1 \geq 0, \ldots, \lambda_4 \geq 0 \begin{cases} a &= \lambda_1 + \lambda_2 \\ b &= \lambda_2 + \lambda_3 - \lambda_4 \\ -a &= -\lambda_2 \\ -b &= -\lambda_3 \\ -1 &\geq -\lambda_3 - \lambda_4 \end{cases}$$

2. Application du lemme de Farkas pour la partie positive :

$$\exists \lambda'_1 \geq 0, \ldots, \lambda'_4 \geq 0 \begin{cases} a &= \lambda'_1 + \lambda'_2 \\ b &= \lambda'_2 + \lambda'_3 - \lambda'_4 \\ 0 &= -\lambda'_2 \\ 0 &= -\lambda'_3 \\ 0 &\geq -\lambda'_3 - \lambda'_4 \end{cases}$$

Le système d'inéquations représentant la boucle est équivalent aux conjonctions de la partie décroissante et de la partie positive :

$$\exists a, b \, \exists \lambda_1 \geq 0, \ldots, \lambda_4 \geq 0, \lambda'_1 \geq 0, \ldots, \lambda'_4 \geq 0 \begin{cases} a &= \lambda_1 + \lambda_2 \\ b &= \lambda_2 + \lambda_3 - \lambda_4 \\ -a &= -\lambda_2 \\ -b &= -\lambda_3 \\ -1 &\geq -\lambda_3 - \lambda_4 \\ a &= \lambda'_1 + \lambda'_2 \\ b &= \lambda'_2 + \lambda'_3 - \lambda'_4 \\ 0 &= -\lambda'_2 \\ 0 &= -\lambda'_3 \\ 0 &\geq -\lambda'_3 - \lambda'_4 \end{cases}$$

En résolvant ce système d'inéquations, on obtient une valeur pour a et une valeur pour b.
On remplace les valeurs de a et b obtenues dans l'équation suivante : $ax + by$ et on obtient la fonction de rang recherchée.
Dans notre cas : $a \geq 1$ et $b = 0$. $\rho(x, y) = x$ est donc une fonction de rang. L'existence de cette fonction de rang prouve la terminaison de notre boucle d'exemple.

S'il n'y avait aucune solution au système d'inéquations, cela aurait signifié que la boucle n'admet pas de fonction de rang linéaire. Souvenons-nous que cela ne nous permet pas de conclure que la boucle ne se termine pas, mais simplement d'affirmer qu'on n'est pas certain qu'elle se termine.

2.2.3 Extension de la méthode

Certaines boucles n'ont pas de fonction de rang linéaire, pourtant elles se terminent. Cela signifie qu'elles ont une fonction de rang non linéaire. La méthode présentée à la section précédente permet seulement de trouver une fonction de rang linéaire pour une boucle LSL.

Cette section présente une extension de cette méthode pour permettre la détection de fonction de rang linéaire à terme. Une telle fonction est une fonction non linéaire qui devient une fonction de rang linéaire, après un certain nombre fini d'itérations de la boucle. On ne sait pas définir après combien d'itérations elle deviendra une simple fonction de rang, mais on peut garantir qu'elle le deviendra.

Pour modéliser le nombre fini d'itérations de la boucle, on suppose que la boucle LSL est donnée avec une fonction linéaire $f(x, y)$ dont la valeur croît à chaque itération de la boucle.

La définition d'une fonction de rang linéaire à terme ρ est donc :

$$\exists k \; \forall x^0, x^1 : \left\{ \begin{array}{l} c(x^0, x^1) \\ f(x^0) \geq k \end{array} \right\} \Longrightarrow \left\{ \begin{array}{l} \rho(x^0) \geq 1 + \rho(x^1) \\ \rho(x^0) \geq 0 \end{array} \right.$$

Trouver une fonction de rang linéaire à terme pour une boucle LSL

Nous allons chercher une fonction de rang linéaire à terme pour une boucle LSL. Présentons tout d'abord, la méthode de manière formelle par l'algorithme 2.3.

Listing 2.3 Méthode pour trouver une fonction de rang linéaire à terme

```
1  input
2          Une boucle LSL \\
3          Une fonction linéaire croissante pour la boucle LSL
4  output
5          Un booléen :
6                  true s'il existe une fonction de rang linéaire à terme
7                  false s'il n'existe pas de fonction de rang linéaire à terme
8  begin
9          Soit ρ(x,y), une fonction de rang de la forme ax + by.
10         DEC(a, k) := résultat de l'application du lemme de Farkas pour la partie négative de ρ
11         DEC1(a), DEC2(a) := linéarisation de DEC(a, k)
12         POS(a, k) := résultat de l'application du lemme de Farkas pour la partie positive de ρ
13         POS1(a), POS2(a) := linéarisation de POS(a, k)
14         if ⋁_{1≤i,j≤2} DECi(a) ∧ POSj(a) est satisfiable
15                 return true
16         else
17                 return false
18 end
```

Pour illustrer cette méthode, nous allons prendre pour exemple la boucle LSL non déterministe suivante :

int x, y
while $(x \geq 0)$
$y \leq y - 1$
$x \leq x + y$

avec la fonction croissante :

$$f(x, y) = -y$$

Il peut paraitre étonnant que cette fonction soit une fonction croissante, pourtant dans le cas de notre exemple c'est bien le cas. En effet, la valeur de la variable y diminuera de minimum une unité (ou restera identique) à chaque itération. Pour un y initial égal à 5, nous aurons par exemple la séquence suivante : 5, 4, 4, 2, 2, 2, 1, -1, -2, -2, ... De ce fait, la valeur de la fonction $f(x, y) = -y$ sera croissante.

Cette boucle n'a pas de fonction de rang linéaire. La première méthode présentée ne pourra donc pas prouver sa terminaison. Nous appliquons donc la méthode pour trouver une fonction de rang linéaire à terme. Commençons par poser le système d'inéquations représentant la boucle. Notons que nous cherchons une fonction de rang linéaire à terme de la forme : $\rho(x, y) = ax + by$. Notre objectif est donc de trouver les valeurs de a et de b.
Voici le système d'inéquations représentant la boucle :

$$\begin{cases} x^0 & \geq & 0 \\ y^1 & \leq & y^0 - 1 \\ x^1 & \leq & x^0 + y^0 \\ -y^0 & \geq & k \end{cases}$$

Pour qu'une fonction de rang linéaire existe, il faut que ce système satisfasse les inéquations suivantes :

$$\begin{cases} ax^0 + by^0 & \geq & 1 + ax^1 + by^1 \\ ax^0 + by^0 & \geq & 0 \end{cases}$$

Pour déduire une fonction de rang satisfaisant le système d'inéquations ci-dessus, nous allons appliquer le lemme de Farkas. Pour faciliter les recherches, nous allons travailler séparément sur la partie décroissante et sur la partie positive.

1. Partie décroissante de la fonction de rang :

$$\exists a, b\ \forall x^0, y^0, x^1, y^1 : \begin{cases} x^0 & \geq & 0 \\ y^1 & \leq & y^0 - 1 \\ x^1 & \leq & x^0 + y^0 \\ -y^0 & \geq & k \end{cases} \Rightarrow ax^0 + by^0 \geq 1 + ax^1 + by^1$$

2. Partie positive de la fonction de rang :

$$\exists a, b\ \forall x^0, y^0, x^1, y^1 : \left\{ \begin{array}{rcl} x^0 & \geq & 0 \\ y^1 & \leq & y^0 - 1 \\ x^1 & \leq & x^0 + y^0 \\ -y^0 & \geq & k \end{array} \right. \Rightarrow ax^0 + by^0 \geq 0$$

Maintenant, nous pouvons préparer l'application du lemme de Farkas. Pour cela, nous allons déduire de chaque partie de la fonction de rang quatre nombres rationnels non négatifs pour la partie décroissante (resp. positive) : $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ (resp. $\lambda'_1, \lambda'_2, \lambda'_3, \lambda'_4$).

1. Préparation de l'application du lemme de Farkas pour la partie décroissante :

$$\begin{array}{rcccccccc} \lambda_1 & : & 1x^0 & +0y^0 & +0x^1 & +0y^1 & +0 & \geq & 0 \\ \lambda_2 & : & 1x^0 & +1y^0 & -1x^1 & +0y^1 & +0 & \geq & 0 \\ \lambda_3 & : & 0x^0 & +1y^0 & +0x^1 & -1y^1 & -1 & \geq & 0 \\ \lambda_4 & : & 0x^0 & -1y^0 & +0x^1 & +0y^1 & -k & \geq & 0 \\ \Rightarrow & & & & & & & & \\ & & ax^0 & +by^0 & -ax^1 & -by^1 & -1 & \geq & 0 \end{array}$$

2. Préparation de l'application du lemme de Farkas pour la partie positive :

$$\begin{array}{rcccccccc} \lambda'_1 & : & 1x^0 & +0y^0 & +0x^1 & +0y^1 & +0 & \geq & 0 \\ \lambda'_2 & : & 1x^0 & +1y^0 & -1x^1 & +0y^1 & +0 & \geq & 0 \\ \lambda'_3 & : & 0x^0 & +1y^0 & +0x^1 & -1y^1 & -1 & \geq & 0 \\ \lambda'_4 & : & 0x^0 & -1y^0 & +0x^1 & +0y^1 & -k & \geq & 0 \\ \Rightarrow & & & & & & & & \\ & & ax^0 & +by^0 & +0x^1 & +0y^1 & +0 & \geq & 0 \end{array}$$

Dès lors que nous avons ces quatre nombres rationnels non négatifs, nous pouvons appliquer le lemme de Farkas sur chacune des parties.

1. Application du lemme de Farkas pour la partie décroissante, on note $DEC(a, b, k)$:

$$\exists \lambda_1 \geq 0, \ldots, \lambda_4 \geq 0 \left\{ \begin{array}{rcl} a & = & \lambda_1 + \lambda_2 \\ b & = & \lambda_2 + \lambda_3 - \lambda_4 \\ -a & = & -\lambda_2 \\ -b & = & -\lambda_3 \\ -1 & \geq & -\lambda_3 - k\lambda_4 \end{array} \right.$$

2. Application du lemme de Farkas pour la partie positive, on note $POS(a, b, k)$:

$$\exists \lambda'_1 \geq 0, \ldots, \lambda'_4 \geq 0 \left\{ \begin{array}{rcl} a & = & \lambda'_1 + \lambda'_2 \\ b & = & \lambda'_2 + \lambda'_3 - \lambda'_4 \\ 0 & = & -\lambda'_2 \\ 0 & = & -\lambda'_3 \\ 0 & \geq & -\lambda'_3 - k\lambda'_4 \end{array} \right.$$

Nous avons donc $DEC(a,b,k)$ et $POS(a,b,k)$. En analysant ces deux systèmes, nous constatons que $k\lambda_4$ et $k\lambda'_4$ introduisent une non-linéarité. Heureusement, nous pouvons l'éliminer grâce aux réécritures suivantes :

$$\begin{array}{rcl}(\lambda_4 \geq 0) & \Leftrightarrow & (\lambda_4 = 0 \vee \lambda_4 > 0 \\ (\lambda'_4 \geq 0) & \Leftrightarrow & (\lambda'_4 = 0 \vee \lambda'_4 > 0)\end{array}$$

Dans le cas où $\lambda_4 > 0$ (resp. $\lambda'_4 > 0$), nous introduisons la variable $P = k\lambda_4$ (resp. $P' = k\lambda'_4$).
Nous obtenons donc les propriétés suivantes :

$$\begin{array}{rcl}\exists k\ DEC(a,b,k) & \Leftrightarrow & DEC_1(a,b) \vee DEC_2(a,b) \\ \exists k\ POS(a,b,k) & \Leftrightarrow & POS_1(a,b) \vee POS_2(a,b)\end{array}$$

Décomposons ces nouveaux systèmes d'inéquations :

1. $$DEC_1(a,b) = \exists \lambda_1 \geq 0, \ldots, \lambda_3 \geq 0 \left\{ \begin{array}{rcl} a & = & \lambda_1 + \lambda_2 \\ b & = & \lambda_2 + \lambda_3 \\ -a & = & -\lambda_2 \\ -b & = & -\lambda_3 \\ -1 & \geq & -\lambda_3 \end{array} \right.$$

2. $$DEC_2(a,b) = \exists \lambda_1 \geq 0, \ldots, \lambda_3 \geq 0, \lambda_4 > 0, P \left\{ \begin{array}{rcl} a & = & \lambda_1 + \lambda_2 \\ b & = & \lambda_2 + \lambda_3 - \lambda_4 \\ -a & = & -\lambda_2 \\ -b & = & -\lambda_3 \\ -1 & \geq & -\lambda_3 - P \end{array} \right.$$

3. $$POS_1(a,b) = \exists \lambda'_1 \geq 0, \ldots, \lambda'_3 \geq 0 \left\{ \begin{array}{rcl} a & = & \lambda'_1 + \lambda'_2 \\ b & = & \lambda'_2 + \lambda'_3 \\ 0 & = & -\lambda'_2 \\ 0 & = & -\lambda'_3 \\ 0 & \geq & -\lambda'_3 \end{array} \right.$$

4. $$POS_2(a,b) = \exists \lambda'_1 \geq 0, \ldots, \lambda'_3 \geq 0, \lambda'_4 > 0, P' \left\{ \begin{array}{rcl} a & = & \lambda'_1 + \lambda'_2 \\ b & = & \lambda'_2 + \lambda'_3 - \lambda'_4 \\ 0 & = & -\lambda'_2 \\ 0 & = & -\lambda'_3 \\ 0 & \geq & -\lambda'_3 - P' \end{array} \right.$$

Pour pouvoir conclure la terminaison de cette boucle, il est nécessaire qu'au moins un de ces systèmes linéaires soit satisfiable :

$$DEC_1(a,b) \wedge POS_1(a,b)$$
$$DEC_1(a,b) \wedge POS_2(a,b)$$
$$DEC_2(a,b) \wedge POS_1(a,b)$$
$$DEC_2(a,b) \wedge POS_2(a,b)$$

Les calculs menant à la vérification de ces systèmes sont volontairement exclus de ce mémoire pour ne pas alourdir inutilement l'exemple. Cependant, après vérification, nous constatons que $DEC_2(a,b) \wedge POS_1(a,b)$ est satisfiable.
Une solution possible de ce système est :

$$b = \lambda_1 = \lambda_3 = \lambda'_2 = \lambda'_3 = 0,$$
$$a = \lambda_2 = \lambda_4 = \lambda'_1 = P = 1$$

En conclusion, $\rho(x,y)$ est une fonction de rang linéaire à terme à partir du seuil $k = \frac{P}{\lambda_4} = 1$. L'existence de la fonction de rang ρ prouve la terminaison de notre boucle d'exemple.

2.3 Méthode sur base de fonction à plusieurs rangs

Une autre méthode pour prouver la terminaison de boucles consiste à se baser sur une fonction à plusieurs rangs, la méthode présentée ici a été développée par Aaron R. Bradley, Zohar Manna et Henny B. Sipma [4].

Cette méthode est une toute autre approche pour prouver la terminaison. Elle se base sur la recherche d'une fonction de rang plus souple, appelée fonction à plusieurs rangs. L'idée d'une telle fonction est d'établir un arbre de fonctions de rang qui prouve la terminaison d'une boucle.

Une fonction à plusieurs rangs ne doit pas décroitre à chaque itération, cependant elle ne peut jamais croitre. De plus, comme une fonction de rang, elle doit avoir une borne inférieure dans la boucle. Les fonctions à plusieurs rangs permettent de prouver la terminaison en prouvant des conditions plus précises. C'est donc une généralisation des fonctions de rang vues dans les sections précédentes.

2.4 Méthode sur base de domaine abstrait

Dans cette section, deux méthodes se basant sur des domaines abstraits et utilisant donc l'interprétation abstraite sont présentées. Ces deux méthodes ont été écrites par Caterina Urban. La première méthode a été publiée dans l'article « Piecewise-Defined Ranking Functions » [20] et la deuxième dans l'article « The Abstract Domain of Segmented Ranking Functions » [21].

2.4.1 Fonction de rang définie par morceaux

Le premier article présenté traite des fonctions de rang définies par morceaux qui ont été développées par Caterina Urban [20].

Dans l'article, elle présente un domaine abstrait pour démontrer la terminaison de programme par l'interprétation abstraite.
Le domaine synthétise automatiquement des fonctions de rang définies par morceaux et déduit des conditions suffisantes pour prouver la terminaison d'un programme.

Pour rappel, un programme se finit lorsqu'il existe une fonction de rang qui décroît durant l'exécution du programme.
Patrick Cousot et Radhia Cousot ont introduit dans leur article [10] l'idée de construire une fonction de rang par interprétation abstraite.

De manière intuitive, nous pouvons concevoir une fonction de rang comme une fonction qui associe chaque état d'un programme à un numéro. On commence par l'état final, la fonction vaut 0. Ensuite, en retraçant le programme à l'envers, on ajoute des états au domaine de la fonction et on compte le nombre de pas effectués comme valeur de la fonction.
Une telle fonction de rang n'est pas calculable, c'est pourquoi, on fait appel à l'interprétation abstraite. Cette dernière permet de générer automatiquement une fonction de rang abstraite qui consiste en des invariants abstraits attachés à des points de programme.
Les éléments du domaine abstrait sont des fonctions définies par morceaux des variables de programme. Ces fonctions représentent une limite supérieure du nombre d'étapes d'exécution du programme qui reste avant la terminaison.

Domaine abstrait de fonctions de rang définies par morceaux

Définition du domaine abstrait $\mathcal{V}^\#$:

$$\mathcal{V}^\# \triangleq \mathcal{S}^\# \mapsto \mathcal{F}^\#$$

$\mathcal{S}^\#$: ensemble d'états abstraits du programme

$\mathcal{F}^\# \triangleq \{\bot_F\} \cup \{f^\# \mid f^\# \in \mathbb{Z}^n \mapsto \mathbb{N}\} \cup \{\top_F\}$
ensemble des valeurs des fonctions de rang des variables du programme

$\bot_F$ représente l'éventuelle non terminaison

$\top_F$ représente le manque d'information pour conclure
Une fonction abstraite $v^\# \in \mathcal{V}^\#$ a la forme :

$$v^\# \equiv \begin{cases} s_1^\# & \mapsto & f_1^\# \\ s_2^\# & \mapsto & f_2^\# \\ \dots & & \\ s_k^\# & \mapsto & f_k^\# \end{cases}$$

où

— Les états abstraits $s_1^\#, \dots, s_k^\#$ introduisent une partition de l'espace des valeurs des variables du programme.

— Les fonctions de rang $f_1^\#, \dots, f_k^\#$ sont des fonctions des variables du programme.

Définition de la fonction de concrétisation γ : Soient $\mathcal{S}$ les états du programme et $\mathbb{O}$ l'ensemble des nombres ordinaux.

$$\gamma \in (\mathcal{S}^\# \mapsto \mathcal{F}^\#) \mapsto (\mathcal{S} \hookrightarrow \mathbb{O})$$

La fonction γ est appliquée par morceaux et fait correspondre une fonction abstraite à une fonction partielle des états du programme à des nombres ordinaux.

$$\begin{array}{lcl} \gamma(s^\# \mapsto \bot_F) & = & \emptyset \\ \gamma(s^\# \mapsto f^\#) & = & \lambda s \in \gamma_s(s^\#).f^\#(s(x_1), \dots, s(x_n)) \\ \gamma(s^\# \mapsto \top_F) & = & \emptyset \end{array}$$

Améliorations : La limite du domaine implémenté amène une imprécision lors de l'analyse des boucles imbriquées et des programmes avec des complexités non linéaires. Pour cette raison, une amélioration serait de concevoir d'autres domaines abstraits, basés sur des abstractions d'états plus sophistiquées et sur des fonctions de rang non linéaires (polynomiales ou exponentielles).

2.4.2 Domaine abstrait de fonctions de rang segmenté

Le deuxième article présenté traite d'un domaine abstrait paramétré pour prouver la terminaison de programme impératif par l'interprétation abstraite. Ce domaine a été développé par Caterina Urban [21].

— Le domaine synthétise automatiquement des fonctions de rang définies par morceaux et déduit des conditions suffisantes pour prouver la terminaison d'un programme.

— Bien que l'analyse effectue des sur-approximations, la solvabilité de la méthode a été prouvée. De ce fait, toutes les exécutions de programme respectant les conditions suffisantes se terminent réellement.

— La segmentation est utilisée pour gérer les disjonctions dans l'analyse des programmes statiques. L'analyse partitionne automatiquement l'espace des valeurs pour les variables du programme par le biais d'un environnement abstrait. Un segment est une paire composée d'un environnement abstrait et d'une fonction abstraite. Durant l'analyse, les segments sont divisés par des tests, modifiés par des affectations et reliés quand ils fusionnent des flux de contrôle.

— Une fonction de rang d'un domaine abstrait segmenté est paramétrée par le choix d'un environnement abstrait (= un intervalle) et le choix d'une fonction abstraite. Cette paramétrisation permet une large gamme d'instanciations du domaine permettant un juste milieu entre la précision de l'analyse et son coût.

Chapitre 3

La méthode de Chen et al.

La méthode de Chen et al. est une méthode qui génère une preuve de terminaison de boucles basée sur des relations de rang linéaires synthétisées. Elle est décrite par Hong Yi Chen, Shaked Flur et Supratik Mukhopadhyay, dans l'article « Termination Proofs for Linear Simple Loops » [7].

Dans ce chapitre, nous présenterons dans l'ordre, une description générale, des définitions utiles pour la compréhension de cette technique, des sous procédures nécessaires pour réaliser cette méthode et enfin son algorithme général.

3.1 Description de la méthode de Chen

La méthode de Chen utilise des travaux de Podelski et Rybalchenko [15], [16]. C'est donc une amélioration de ces travaux. C'est aussi une amélioration par rapport aux techniques basées sur des fonctions à plusieurs rangs, décrite à la section 2.3.

Avec cette méthode, on est uniquement capable de déterminer les boucles terminantes. L'algorithme est donc borné. Si une preuve de terminaison n'a toujours pas été trouvée après un certain nombre d'itérations, il s'arrête. Pour trouver une preuve de terminaison, l'espace d'états du programme est répétitivement séparé en différents sous-espaces. Des fonctions de rang linéaires seront cherchées sur chacun de ces sous-espaces. Ensuite, il faudra vérifier que la fermeture transitive de la relation de transition du programme soit inclue dans l'union des relations de rang donc des fonctions de rang linéaires trouvées. Soient R^+ la fermeture transitive de la relation de transition de la boucle L et T l'union des relations de rang, alors l'inclusion à vérifier est $R^+(L) \subseteq T$.

La signification des termes « relation de rang » et « fermeture transitive » sera formellement définie à la section 3.2.

Dans cette section la terminaison est celle définie dans l'article [16] qui se base sur le théorème de Ramsey décrit dans son article [17] :

Un programme P se termine si et seulement si il existe une relation de rang disjonctive T telle que $R^+ \subseteq T$.

L'algorithme de la méthode est décrit à la section 3.6. Cet algorithme nécessite des étapes intermédiaires :

1. Construire une relation de rang disjonctive T.
   Pour cela, plusieurs étapes sont effectuées :
   (a) Construction d'une simple fonction de rang.
   Cette étape est décrite aux sections 2.1 et 3.3.
   (b) Séparation de l'espace d'états lorsqu'une simple fonction de rang ne peut pas être construite directement.
   Cette étape est décrite à la section 3.4
2. Vérifier que la fermeture transitive du programme est un sous-ensemble de la relation de rang disjonctive ($R^+ \subseteq T$).
   Cette étape est décrite à la section 3.5.

3.2 Définitions

3.2.1 Ensemble bien fondé

Un ensemble D est un ensemble bien fondé selon une relation linéaire $\leq$ s'il n'y a pas de séquence décroissante infinie $d_0 > d_1 > d_2 > \ldots$ d'éléments dans D.

3.2.2 Fonction de rang

Selon une relation de transition $R \subseteq S \times S$, une fonction $r : S \to D$ est une fonction de rang pour R, si D est un ensemble bien fondé et que pour chaque $(s_1, s_2) \in R$ nous avons $r(s_1) > r(s_2)$.

3.2.3 Relation de rang

Soit une fonction de rang : $r : S \to D$ (avec une relation de transition R), on définit une relation de rang correspondante sur S par :

$$\tau(r) = \{(s_1, s_2) \in S \times S \mid r(s_1) > r(s_2)\}.$$

3.2.4 Fermeture transitive

Soit R une relation binaire, sa fermeture transitive est :

$$\begin{array}{rcl} R^+ & = & \bigcup_{n\in\mathbb{N}} R^n \\ R^n & = & R \times R \times \cdots \times R \\ & = & \text{Produit cartésien composé de } n \text{ fois } R \end{array}$$

En pratique, pour trouver la fermeture transitive d'un système de transition, on procède comme suit : s'il y a un chemin entre a et b, on ajoute un arc entre a et b. En d'autres termes, il faut trouver tous les arcs transitifs.

3.2.5 Relation de rang disjonctive

Soient $T_1, \ldots, T_n$ des relations de rang.
Si

$$T = T_1 \cup \cdots \cup T_n$$

alors, T est une relation de rang disjonctive.

3.2.6 Représentation de LSL

Dans cette méthode, on représente une boucle linéaire simple, LSL, sous la forme suivante :

$$L = \langle \textsc{Cond}, \textsc{Update}, i, j \rangle$$

— $X^0 = (x_1, x_2, \ldots, x_n)$. Ce sont les variables de la boucle L ;

— $X^1, X^2, \ldots, X^n (m, n \geq 1)$ sont les copies des variables de X à chaque itération n ;

— Cond est un ensemble de contraintes linéaires de la forme :
$a_i X^i \bowtie b$, avec $\bowtie \in \{<, \leq, =, \geq, >\}$
Il représente l'ensemble des inéquations (ou équations) présentes dans la condition permettant d'entrer dans le corps de la boucle.

— Update est un ensemble de contraintes linéaires de la forme :
$a_0 X^0 + \cdots + a_n X^n \bowtie b$, avec $\bowtie \in \{<, \leq, =, \geq, >\}$
Il représente l'ensemble des inéquations (ou équations) du corps de la boucle.

— n est le nombre final d'itérations ;

— i et j sont des entiers et $0 \leq i \leq j \leq n$;

- i représente le nombre d'itérations de la boucle déjà effectué avant l'itération courante ;
- j représente le nombre d'itérations de la boucle effectué après l'itération courante ;
- Par défaut, on admet que $i = 0$ et que $j = 1$.

— a_k et b sont des coefficients sur $\mathbb{Z}$ ou $\mathbb{Q}$.

3.3 Méthode étendue pour les fonctions de rang linéaires

L'algorithme de Podelski & Rybalchenko qui a été présenté à la section 2.1 ne gère que les transitions entre X^0 et X^1. La méthode étudiée ici a développé un nouvel algorithme (voir 3.1). Cet algorithme peut gérer les transitions sur n'importe quel nombre de copies de X^0.

Quelques notations sont nécessaires :

— $\mathbf{A}$ indique le vecteur $(A^0 \dots A^i \dots A^j \dots A^n)$

— $\mathbf{A_i}$ indique l'élément i : A^i

— $\mathbf{A_{-i}}$ indique le vecteur sans l'élément i : $(A^0 \dots A^{i-1} A^{i+1} \dots A^j \dots A^n)$

Listing 3.1 Méthode étendue pour les fonctions de rang linéaires

input
    programme $\mathbf{AX} \leq b$
output
    Un booléen :
        true s'il existe une fonction de rang linéaire
        false s'il n'existe pas de fonction de rang linéaire
begin
    if $\exists \lambda_1, \lambda_2 \in \mathbb{Q}$:

$$\begin{cases} \lambda_1 \mathbf{A}_{-i} & = & 0 \\ \lambda_2 \mathbf{A}_{-i,-j} & = & 0 \\ (\lambda_1 - \lambda_2)\mathbf{A}_i & = & 0 \\ \lambda_2(\mathbf{A}_i + \mathbf{A}_j) & = & 0 \\ \lambda_2 b & < & 0 \end{cases}$$

    then

$$\rho(X) = \begin{cases} \lambda_2 \mathbf{A}_j X^i & \text{s'il existe } X^j \text{ tel que } (X^i, X^j) \in R(L) \\ (-\lambda_1 b) - (-\lambda_2 b) & \text{si non} \end{cases}$$

        return true
    else
        return false
end.

3.4 Séparer l'espace d'états

Dans le cas où une fonction de rang linéaire ne peut pas être directement trouvée par la méthode de Podelski & Rybalchenko (section 2.1) ou par la méthode étendue (section 3.3), il faut séparer l'espace d'états. Cette séparation se fait suivant la méthode *diviser pour régner*.

L'explication de cette méthode se trouve à la section suivante (section 3.4.1). Pour simplifier cette explication, deux procédures sont définies. La

première procédure, appelée SHIFT, a pour objectif de transformer les contraintes d'une itération de boucle à l'itération suivante. La deuxième procédure, appelée DIFF, a pour objectif de créer une nouvelle contrainte qui sera utilisée pour partitionner l'espace d'états. Ces deux procédures seront plus détaillées respectivement aux sections 3.4.2 et 3.4.3.

Enfin, une application de cette méthode sera présentée à la section 3.4.5.

3.4.1 Méthode *diviser pour régner*

La méthode *diviser pour régner* se déroule en trois étapes.

Premièrement, on formera un sous-ensemble composé d'une ou plusieurs inéquations (ou équations) de la condition ou du corps de la boucle. Ces inéquations (ou équations) sont choisies arbitrairement.

Deuxièmement, chaque inéquation (ou équation) du sous-ensemble formé sera considérée comme un sous-ensemble.

Enfin, troisièmement, l'union de tous les sous-ensembles de chacune des inéquations (ou équations) sera aussi considérée comme un sous-ensemble.

Dans la description formelle des trois étapes à suivre, nous utilisons quelques abréviations :

- SEED représente le sous-ensemble sélectionné à partir de toutes les inéquations de COND et de UPDATE ;
- φ représente une inéquation (ou équation) ;
- L_{triv} représente une nouvelle boucle dérivée à partir d'une seule inéquation (ou équation) de SEED ;
- L_{synth} représente une nouvelle boucle dérivée à partir de l'ensemble des inéquations (ou équations) de SEED ;
- SEED_{triv} (resp. SEED_{synth}) sont des variables temporaires pour calculer L_{triv} (resp. L_{synth}).

Voici les trois étapes de la méthode :

1. Soit $L = \langle \text{COND}, \text{UPDATE}, i, j \rangle$
   Soit SEED un sous-ensemble de $\text{COND} \cup \text{UPDATE}$

2. Pour chaque $\varphi \in \text{SEED}$:

$$\begin{aligned} \text{SEED}_{triv} &\leftarrow \{\text{DIFF}_{j,>}(\varphi)\} \\ L_{triv} &\leftarrow \langle \text{COND}, \text{UPDATE} \cup \text{SEED}_{triv}, i, j \rangle \end{aligned}$$

3. Calculer L_{synth} :

$$\begin{aligned} \text{SEED}_{synth} &\leftarrow \text{DIFF}_{j,\leq}(\text{SEED}) \\ L_{synth} &\leftarrow \langle \text{COND}, \text{UPDATE} \cup \text{SEED}_{synth}, i, j \rangle \end{aligned}$$

Concrètement, si SEED ne contient qu'une seule inéquation ($|\text{SEED}| = 1$), alors la boucle donnera lieu à deux boucles simplifiées. Nous pouvons généraliser cela en disant que si SEED contient n inéquations ($|\text{SEED}| = n$), alors chaque boucle donnera lieu à $n + 1$ boucles simplifiées.

Il est de pratique courante de prendre comme SEED, pour la première itération, les conditions à satisfaire avant d'évaluer le corps de la boucle, c'est-à-dire COND. Ensuite, pour les itérations suivantes, il est courant de prendre comme SEED, les contraintes qui ont été ajoutées pour définir le cas synthétisé de l'itération précédente.

3.4.2 Procédure Shift

SHIFT_i est un processus qui transforme des contraintes d'un état X à un état X^i supérieur. Par exemple, $\text{SHIFT}_1(x^0 - x^1 < 1) = x^1 - x^2 < 1$. En d'autres termes, chaque variable est augmentée de i degré, ce qui représente donc l'itération suivante.

3.4.3 Procédure Diff

— Soit une contrainte linéaire $\varphi : \psi \leq b$ et un ensemble de contraintes SEED, où $b \in \mathbb{Z}$. On définit :

$$\begin{aligned} \text{DIFF}_{i,\sim}(\varphi) &\triangleq \text{SHIFT}_i(\psi) \sim \psi \\ \text{DIFF}_{i,\sim}(\text{SEED}) &\triangleq \{\text{DIFF}_{i,\sim}(\varphi) \mid \varphi \in \text{SEED}\} \end{aligned}$$

où $\sim \in \{>, \leq\}$.

— Soit une contrainte linéaire $\varphi : \psi \geq b$ et un ensemble de contraintes SEED, où $b \in \mathbb{Z}$. On définit :

$$\begin{aligned} \text{DIFF}_{i,\sim}(\varphi) &\triangleq \psi \sim \text{SHIFT}_i(\psi) \\ \text{DIFF}_{i,\sim}(\text{SEED}) &\triangleq \{\text{DIFF}_{i,\sim}(\varphi) \mid \varphi \in \text{SEED}\} \end{aligned}$$

où $\sim \in \{>, \leq\}$.

3.4.4 Variables réelles ou rationnelles

Lorsque les variables sont des nombres réels ou rationnels ($\mathbb{R}$ ou $\mathbb{Q}$), les contraintes $\varphi \geq b$ et $\varphi > \text{SHIFT}(\varphi)$ ne garantissent plus que φ est une fonction linéaire. Heureusement, il existe deux solutions pour remédier à cela :

— Prendre une petite valeur positive c et partitionner l'espace d'états par $\varphi - \text{SHIFT}(\varphi) > c$ et sa négation $\varphi - \text{SHIFT}(\varphi) \leq c$, respectivement le cas trivial et le cas synthétisé.

— Partitionner de la même manière que pour des entiers, avec $\varphi \geq b$ et $\varphi > \text{SHIFT}(\varphi)$. Cependant, il n'y aura plus de cas trivial, donc il faut continuer de partitionner sur les deux nouveaux cas générés.

3.4.5 Exemple d'application : *diviser pour régner*

Nous allons voir un exemple concret d'application de la méthode *diviser pour régner*. Cela a pour objectif de mieux comprendre comment utiliser les procédures SHIFT et DIFF.

Pour notre exemple, nous utilisons la boucle LSL suivante :

while $(x \geq 0)$
$\quad x := x - 1$

Ce même programme peut s'exprimer sous la forme :

$$L = \langle \textsc{Cond}, \textsc{Update}, i, j \rangle.$$

Ce qui nous donne :

$$L = \langle \{x^0 \geq 0\}, \{x^1 = x^0 - 1\}, 0, 1 \rangle$$

Pour commencer, nous allons définir notre SEED. Nous allons suivre la pratique courante et prendre comme SEED, la condition de la boucle. Nous obtenons donc :

$$\textsc{Seed} = \{x^0 \geq 0\}$$

Maintenant que nous avons défini notre SEED, nous avons pour objectif premier de définir L_{triv}. Pour cela, nous devons choisir une inéquation, φ, dans notre ensemble SEED. Étant donné que notre ensemble SEED ne contient qu'une seule inéquation, il va de soi que nous avons :

$$\varphi : x^0 \geq 0$$

Ensuite, pour calculer L_{triv}, nous devons tout d'abord calculer $\textsc{Seed}_{triv}$. Comme le définit la méthode, $\textsc{Seed}_{triv}$ s'obtient en appliquant la méthode $\textsc{Diff}_{j,>}$ sur notre inéquation φ. Nous calculons donc :

$$\textsc{Seed}_{triv} = \textsc{Diff}_{1,>}(\varphi),$$

ce qui nous donne :

$$\textsc{Seed}_{triv} = x^0 > x^1.$$

Nous sommes maintenant prêts à calculer L_{triv}. Pour rappel L_{triv} s'obtient par la formule suivante :

$$L_{triv} \leftarrow \langle \textsc{Cond}, \textsc{Update} \cup \textsc{Seed}_{triv}, i, j \rangle,$$

nous obtenons donc :

$$L_{triv} = \langle \{x^0 \geq 0\}, \{x^1 = x^0 - 1 \cup x^0 > x^1\}, 0, 1 \rangle.$$

Nous devons bien évidemment aussi calculer L_{synth}. Pour cela, nous devons tout d'abord calculer $\textsc{Seed}_{synth}$. Comme le définit la méthode, $\textsc{Seed}_{synth}$ s'obtient en appliquant la méthode $\textsc{Diff}_{j,\leq}$ sur l'ensemble $\textsc{Seed}$. Nous calculons donc :

$$\textsc{Diff}_{1,\leq}(\textsc{Seed}),$$

ce qui nous donne :

$$\textsc{Seed}_{synth} = x^0 \leq x^1.$$

Nous sommes maintenant prêts à calculer L_{synth}. Pour rappel L_{synth} s'obtient par la formule suivante :

$$L_{synth} \leftarrow \langle \textsc{Cond}, \textsc{Update} \cup \textsc{Seed}_{synth}, i, j \rangle,$$

nous obtenons donc :

$$L_{synth} = \langle \{x^0 \geq 0\}, \{x^1 = x^0 - 1 \cup x^0 \leq x^1\}, 0, 1 \rangle.$$

En conclusion de cet exemple, nous avons séparé notre boucle initiale $L = \langle \{x^0 \geq 0\}, \{x^1 = x^0 - 1\}, 0, 1 \rangle$ en deux boucles L_{triv} et L_{synth} :

$$\begin{array}{rcl} L_{triv} & = & \langle \{x^0 \geq 0\}, \{x^1 = x^0 - 1 \cup x^0 > x^1\}, 0, 1 \rangle \\ L_{synth} & = & \langle \{x^0 \geq 0\}, \{x^1 = x^0 - 1 \cup x^0 \leq x^1\}, 0, 1 \rangle \end{array}$$

3.5 Vérifier l'inclusion de R^+ dans T

Dans cette section, nous allons détailler comment vérifier l'inclusion de R^+ dans T.
Pour rappel R^+ représente la fermeture transitive de la relation de transition et T représente l'union des relations de rang. Supposons que nous avons les relations de rang suivantes : $\tau_1, \tau_2, \ldots, \tau_n$, alors, T sera équivalent à $\tau_1 \cup \tau_2 \cup \cdots \cup \tau_n$.

Grâce à l'article « Transition Invariants » [16], nous savons que $R \subseteq T \wedge (T \circ R \subseteq T \vee R \circ T \subseteq T)$ est une condition suffisante pour démontrer $R^+ \subseteq T$.

$$R \subseteq T \wedge (T \circ R \subseteq T \vee R \circ T \subseteq T) \Rightarrow R^+ \subseteq T \tag{3.1}$$

Il nous suffit donc de démontrer que $R \subseteq T \wedge (T \circ R \subseteq T \vee R \circ T \subseteq T)$ est vrai. Dans le but d'automatiser le test de cette condition, nous allons utiliser le solveur de satisfiabilité Z3 [1]. Pour cela, il faut exprimer la condition sous forme de question existentielle.
Nous considérons $R \subseteq T$ comme le cas de base de la démonstration et $(T \circ R \subseteq T \vee R \circ T \subseteq T)$ comme le cas inductif.

3.5.1 Cas de base de la vérification

T est une disjonction de n fonctions de rang. On peut donc remplacer T par $\tau_1 \cup \cdots \cup \tau_n$.

$$R \subseteq T \quad = \quad R \subseteq \tau_1 \cup \cdots \cup \tau_n \tag{3.2}$$

R est inclus dans la disjonction des fonctions de rang, si et seulement si, chaque contrainte de R est une condition nécessaire pour la disjonction des fonctions de rang. x représente les variables dans les contraintes, c représente les contraintes de R. L'équation 3.2 peut se réécrire sous la forme :

$$\forall x(c \Rightarrow \tau_1 \vee \cdots \vee \tau_n) \tag{3.3}$$

Pour pouvoir utiliser un solveur existentiel, il faut transformer le quantificateur universel en un quantificateur existentiel. La double négation permet cela. L'équation 3.3 devient :

$$\neg\neg\forall x(c \Rightarrow \tau_1 \vee \cdots \vee \tau_n) \tag{3.4}$$

Ensuite, dans le but d'obtenir une forme conjonctive normale, on transforme l'implication en disjonction par la règle : $a \Rightarrow b \equiv \neg a \vee b$. L'équation 3.4 devient :

$$\neg\neg\forall x(\neg c \vee \tau_1 \vee \cdots \vee \tau_n) \tag{3.5}$$

Il ne nous reste plus qu'à appliquer une négation, pour obtenir le quantificateur existentiel. L'équation 3.5 devient :

$$\neg(\exists x(c \wedge \neg\tau_1 \wedge \cdots \wedge \neg\tau_n)) \tag{3.6}$$

Cette dernière formule peut facilement être testée automatiquement par un solveur de satisfiabilité.

3.5.2 Cas inductif de la vérification

Le même processus va être appliqué aux formules $T \circ R \subseteq T$ et $R \circ T \subseteq T$. On remplace d'abord T par sa disjonction de n fonctions de rang (équations 3.7 et 3.12). On transforme la contrainte d'inclusion sous forme d'implication (équations 3.8 et 3.13). On rajoute une double négation, dans le but d'obtenir un quantificateur existentiel à la place du quantificateur universel (équations 3.9 et 3.14). Dans le but d'obtenir une forme conjonctive normale, on transforme l'implication en disjonction (équations 3.10 et 3.15). Enfin, on applique une négation pour obtenir le quantificateur existentiel (équations 3.11 et 3.16). Ces deux dernières équations peuvent facilement être testées par un solveur de satisfiabilité.

1.

$$\begin{aligned}
T \circ R \subseteq T &= (\tau_1 \circ R) \vee \cdots \vee (\tau_n \circ R) \subseteq \tau_1 \vee \cdots \vee \tau_n && (3.7)\\
&= \forall x[((\tau_1 \circ c) \vee \cdots \vee (\tau_n \circ c)) \Rightarrow \tau_1 \vee \cdots \vee \tau_n] && (3.8)\\
&= \neg\neg\forall x[((\tau_1 \circ c) \vee \cdots \vee (\tau_n \circ c)) \Rightarrow \tau_1 \vee \cdots \vee \tau_n] && (3.9)\\
&= \neg\neg\forall x[\neg[(\tau_1 \circ c) \vee \cdots \vee (\tau_n \circ c)] \vee \tau_1 \vee \cdots \vee \tau_n] && (3.10)\\
&= \neg(\exists x[((\tau_1 \circ c) \vee \cdots \vee (\tau_n \circ c)) \wedge \neg\tau_1 \wedge \cdots \wedge \neg\tau_n] && (3.11)
\end{aligned}$$

2.

$$\begin{aligned}
R \circ T \subseteq T &= (R \circ \tau_1) \vee \cdots \vee (R \circ \tau_n) \subseteq \tau_1 \vee \cdots \vee \tau_n && (3.12)\\
&= \forall x[((c \circ \tau_1) \vee \cdots \vee (c \circ \tau_n)) \Rightarrow \tau_1 \vee \cdots \vee \tau_n] && (3.13)\\
&= \neg\neg\forall x[((c \circ \tau_1) \vee \cdots \vee (c \circ \tau_n)) \Rightarrow \tau_1 \vee \cdots \vee \tau_n] && (3.14)\\
&= \neg\neg\forall x[\neg[(c \circ \tau_1) \vee \cdots \vee (c \circ \tau_n)] \vee \tau_1 \vee \cdots \vee \tau_n] && (3.15)\\
&= \neg(\exists x[((c \circ \tau_1) \vee \cdots \vee (c \circ \tau_n)) \wedge \neg\tau_1 \wedge \cdots \wedge \neg\tau_n] && (3.16)
\end{aligned}$$

3.5.3 Définition d'une composition relationnelle

Voici la définition de la composition relationnelle utilisée dans les exemples :

$$T \circ R = \{(x,t) \mid \exists y,u : (x,y) \in R \wedge (u,t) \in T \wedge y = u\}$$

3.6 Algorithme général de la méthode de Chen

Maintenant que tous les concepts sous-jacents ont été définis, voici de façon non formelle la définition de l'algorithme de la méthode de Chen.

Soit une boucle L :

1. Traduire la boucle sous la forme : $L = \langle \textsc{Cond}, \textsc{Update}, i, j \rangle$
2. Détecter par la méthode de Podelski &Rybalchenko si L a une fonction de rang linéaire.

   Si oui : la fonction de rang linéaire est retournée, le programme se termine

   Si non : on sépare L par la méthode « diviser pour régner »

   - *Cas trivial* : on déduit une fonction de rang linéaire par cas trivial
   - *Cas synthétisé* : on détecte par la méthode de Podelski & Rybalchenko si ce cas a une fonction de rang linéaire (*) :

     Si oui : on teste si l'union des fonctions de rang linéaires trouvée, T, satisfont : $R^+(L) \subseteq T$

     Si oui : le programme se termine

     Si non : on cherche un chemin d'erreur L' qui exécute un certain nombre de fois la boucle L
     on détecte par la méthode de Podelski & Rybalchenko une fonction de rang linéaire de L'
     on retourne à (*)

     Si non : on sépare le cas synthétisé par la méthode « diviser pour régner »

     - *Cas trivial* : on déduit une fonction de rang linéaire par cas trivial
     - *Cas synthétisé* : on détecte par la méthode de Podelski & Rybalchenko étendue si ce cas a une fonction de rang linéaire
       on retourne à (*)

Chapitre 4

Mise en pratique de la méthode de Chen et al.

Pour illustrer la méthode de Chen et al. décrite à la section précédente, plusieurs exemples ont été développés. Dans ces exemples, on dit qu'un programme est linéaire si et seulement s'il existe une fonction de rang linéaire pour ce programme. Similairement, on dit qu'un programme est non linéaire si et seulement s'il n'existe pas de fonction de rang linéaire pour ce programme.

— Le premier exemple est un programme non linéaire se terminant. La méthode de Chen n'est pas capable de le déterminer.

— Le deuxième est un programme linéaire se terminant. La méthode de Chen est capable de le déterminer.

— Le troisième est un programme non linéaire se terminant. La méthode de Chen est capable de le déterminer.

— Le quatrième est également un programme non linéaire se terminant. La méthode de Chen est capable de le déterminer.

La méthode de Chen est complète pour les programmes linéaires. Il est donc impossible de montrer un exemple de programme linéaire se terminant que la méthode de Chen ne soit pas capable de déterminer.

Toutes les variables de ces exemples prennent leurs valeurs dans $\mathbb{Z}$. Étant donné que les nombres relatifs sont un sous-ensemble des nombres rationnels, si un programme se termine dans $\mathbb{Q}$, il se terminera aussi dans $\mathbb{Z}$.

Pour réaliser ces exemples, plusieurs outils informatiques ont été utilisés :

— Le langage de programmation *Scheme* pour les méthodes de Podelski & Rybalchenko normales et étendues.
Sur base du système d'inéquations exprimé sous la forme matricielle $(A^0 A^1)X \leq b$, le code source (à l'annexe A.1) permet de générer le

système d'inéquations :

$$\begin{cases} \lambda_1, \lambda_2 & \geq & 0 \\ \lambda_1 A^1 & = & 0 \\ (\lambda_1 - \lambda_2) A^0 & = & 0 \\ \lambda_2 (A^0 + A^1) & = & 0 \\ \lambda_2 b & < & 0 \end{cases}$$

Ces systèmes d'inéquations générés se trouvent à l'annexe A.2.

— Les systèmes générés de l'annexe A.2 ont été remis en forme à la main pour en faciliter la lecture. La résolution de ces systèmes s'est faite en *SWI-Prolog*. Le code source utilisé se trouve à l'annexe A.3.

— Les tests de satisfiabilité pour démontrer $R \subseteq T \wedge (T \circ R \subseteq T \vee R \circ T \subseteq T)$ ont été soumis au solveur *Z3*. Le code source utilisé se trouve à l'annexe A.4. Les résultats obtenus se situent à l'annexe A.5.

4.1 Premier exemple

Comme programme non linéaire se terminant, nous prenons pour exemple la boucle $n^o\,1$ de la suite de tests de Chen [6].

```
int x
while (x > 0)
        x := -2x + 10
```

$$L \triangleq \langle \{x^0 > 0\}, \{x^1 = -2x^0 + 10\}, 0, 1 \rangle$$

Nous constatons que L n'est pas linéaire, donc nous séparons notre boucle LSL en suivant la méthode *diviser pour régner*.
Comme la pratique courante le veut, nous utilisons $\textsc{Cond}_L$ comme Seed. Nous obtenons alors :

$$\begin{array}{lcll} L_{1.1} & = & \langle \textsc{Cond}_L, \textsc{Update}_L \cup \{x^0 > x^1\}, 0, 1 \rangle & \text{(cas trivial)} \\ L_{1.2} & = & \langle \textsc{Cond}_L, \textsc{Update}_L \cup \{x^0 \leq x^1\}, 0, 1 \rangle & \text{(cas synthétisé)} \end{array}$$

Analysons les différents cas :

Cas trivial : x^1 est inférieur à x^0 et on a une borne inférieure, on a donc une fonction de rang linéaire : $\rho_1(X^0) = x^0$

Cas synthétisé : on teste par la méthode de Podelski & Rybalchenko si on a une fonction de rang linéaire :

1. La première étape consiste à transformer la boucle $L_{1.2}$ en système d'inéquations :

$$\begin{cases} x^0 & > & 0 \\ x^1 & \leq & -2x^0 + 10 \\ x^1 & \geq & -2x^0 + 10 \\ x^0 & \leq & x^1 \end{cases} \Leftrightarrow \begin{cases} -x^0 & \leq & 0 \\ x^1 + 2x^0 & \leq & 10 \\ -x^1 - 2x^0 & \leq & -10 \\ x^0 - x^1 & \leq & 0 \end{cases}$$

2. Ensuite, écrivons le système sous forme matricielle $(A^0 A^1)X \leq b$:

$$\begin{pmatrix} -1 & 0 \\ 2 & 1 \\ -2 & -1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x^0 \\ x^1 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 10 \\ -10 \\ 0 \end{pmatrix}$$

3. Nous pouvons dès lors résoudre le système d'inéquations en posant $\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3, \lambda'_4)$ et $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3, \lambda''_4)$:

$$\begin{cases} \lambda_1, \lambda_2 & \geq & 0 \\ \lambda_1 A^1 & = & 0 \\ (\lambda_1 - \lambda_2)A^0 & = & 0 \\ \lambda_2(A^0 + A^1) & = & 0 \\ \lambda_2 b & < & 0 \end{cases} \rightarrow \begin{cases} \lambda_1, \lambda_2 \geq 0 \\ \lambda'_2 - \lambda'_3 - \lambda'_4 = 0 \\ 2\lambda'_2 - \lambda'_1 - \lambda''_1 - 2\lambda''_2 \\ \quad -2\lambda'_3 + 2\lambda''_3 + \lambda'_4 - \lambda''_4 = 0 \\ -\lambda''_1 + 3\lambda''_2 - 3\lambda''_3 = 0 \\ 10\lambda''_2 - 10\lambda''_3 < 0 \end{cases}$$

Nous constatons que ce système n'a pas de solution. Il n'existe donc pas de fonction de rang linéaire correspondant à ce système d'inéquations. Dans le but de tout de même trouver une fonction de rang correspondant à notre programme, nous allons dérouler notre boucle L d'une itération supplémentaire. Ensuite, nous pourrons reséparer cette boucle en plusieurs sous-boucles via la méthode *diviser pour régner*.
L' correspond à la boucle L déroulée d'une itération supplémentaire et L_2 correspond à notre nouvelle boucle, c'est-à-dire la boucle déroulée L' à laquelle nous avons rajouté notre SEED précédent comme inéquation du corps de la boucle.
Nous obtenons donc :

$$\begin{aligned} L' &= \langle \{x^0 \geq 0\}, \\ &\quad \{x^1 = -2x^0 + 10, \\ &\quad x^1 \geq 0, x^2 = -2x^1 + 10\}, \\ &\quad 0, 1\rangle \\ L_2 &= \langle \text{COND}_L, \text{UPDATE}_{L'} \cup \text{SEED}, 0, 1\rangle \\ &= \langle \{x^0 \geq 0\}, \\ &\quad \{x^1 = -2x^0 + 10, \\ &\quad x^1 \geq 0, x^2 = -2x^1 + 10\} \cup \{x^1 \geq x^0\}, \\ &\quad 0, 1\rangle \end{aligned}$$

Nous constatons que L_2 n'est pas linéaire, donc nous séparons notre boucle LSL L_2 en suivant la méthode *diviser pour régner*.
Comme la pratique courante le veut, nous utilisons notre SEED_L précédent ($= x^1 \geq x^0$) comme SEED actuel. Nous obtenons alors :

$$\begin{aligned} L_{2.1} &= \langle \text{COND}_{L_2}, \text{UPDATE}_{L_2} \cup \{(x^1 - x^0) > (x^2 - x^1)\}, 0, 1\rangle \quad \text{(cas trivial)} \\ L_{2.2} &= \langle \text{COND}_{L_2}, \text{UPDATE}_{L_2} \cup \{(x^1 - x^0) \leq (x^2 - x^1)\}, 0, 1\rangle \quad \text{(cas synthétisé)} \end{aligned}$$

Analysons les différents cas :

Cas trivial : $L_{2.1}$ est un cas trivial où une fonction de rang linéaire est garantie :
$\rho_2(X^0) = x^1 - x^0$

Cas synthétisé : on ne peut pas appliquer la méthode classique pour obtenir une fonction de rang linéaire, étant donné que $L_{2.2}$ implique X^2. On applique donc la méthode étendue :

1. La première étape consiste à transformer la boucle $L_{2.2}$ en système d'inéquations :

$$\begin{cases} x^0 & > & 0 \\ x^1 & \leq & -2x^0 + 10 \\ x^1 & \geq & -2x^0 + 10 \\ x^1 & \geq & 0 \\ x^2 & \leq & -2x^1 + 10 \\ x^2 & \geq & -2x^1 + 10 \\ x^1 & \geq & x^0 \\ (x^1 - x^0) & \leq & (x^2 - x^1) \end{cases} \Leftrightarrow \begin{cases} -x^0 & \leq & 0 \\ x^1 + 2x^0 & \leq & 10 \\ -x^1 - 2x^0 & \leq & -10 \\ -x^1 & \leq & 0 \\ x^2 + 2x^1 & \leq & 10 \\ -x^2 - 2x^1 & \leq & -10 \\ x^0 - x^1 & \leq & 0 \\ -x^0 + 2x^1 - x^2 & \leq & 0 \end{cases}$$

2. Ensuite, écrivons le système sous forme matricielle $(A^0 A^1)X \leq b$:

$$\begin{pmatrix} -1 & 0 & 0 \\ 2 & 1 & 0 \\ -2 & -1 & 0 \\ 0 & -1 & 0 \\ 0 & 2 & 1 \\ 0 & -2 & -1 \\ 1 & -1 & 0 \\ 0 & 2 & -1 \end{pmatrix} X \leq \begin{pmatrix} 0 \\ 10 \\ -10 \\ 0 \\ 10 \\ -10 \\ 0 \\ 0 \end{pmatrix}$$

3. Nous pouvons dès lors résoudre le système d'inéquations en posant
$\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3, \lambda'_4, \lambda'_5, \lambda'_6, \lambda'_7, \lambda'_8)$ et $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3, \lambda_4, \lambda''_5, \lambda''_6, \lambda''_7, \lambda''_8)$:

$$\begin{cases} \lambda_1 \mathbf{A}_{-i} & = & 0 \\ \lambda_2 \mathbf{A}_{-i,-j} & = & 0 \\ (\lambda_1 - \lambda_2)\mathbf{A}_i & = & 0 \\ \lambda_2(\mathbf{A}_i + \mathbf{A}_j) & = & 0 \\ \lambda_2 b & < & 0 \end{cases} \rightarrow \begin{cases} \lambda'_2 - \lambda'_3 - \lambda'_4 + 2\lambda'_5 - 2\lambda'_6 - \lambda'_7 + 2\lambda'_8 = 0 \\ \lambda'_5 - \lambda'_6 - \lambda'_8 = 0 \\ \lambda''_5 = \lambda''_6 - \lambda''_8 = 0 \\ -\lambda'_1 + \lambda''_1 + 2\lambda'_2 - 2\lambda''_2 \\ \quad -2\lambda'_3 + 2\lambda''_3 + \lambda'_7 - \lambda''_7 = 0 \\ -\lambda''_1 + 3\lambda''_2 - 3\lambda''_3 - \lambda''_4 \\ \quad +2\lambda''_5 - 2\lambda''_6 + 2\lambda''_8 = 0 \\ 10\lambda''_2 - 10\lambda''_3 + 10\lambda''_5 - 10\lambda''_6 < 0 \end{cases}$$

Nous constatons que ce système n'a pas de solution. Il n'existe donc pas de fonction de rang linéaire correspondant à ce système

d'inéquations. De plus, selon l'article [7], à la section 5, il est stipulé que la terminaison de ce programme n'est pas détectée par la méthode appliquée.
Continuer à séparer l'espace d'états, n'amènerait donc jamais à une preuve de terminaison.

Conclusion : Malgré que le programme se termine, cette méthode n'est pas capable de le démontrer. Ceci s'explique simplement : ce programme se termine pour les entrées appartenant à $\mathbb{Z}$, or la méthode de Podelski & Rybalchenko trouve une fonction de rang dans $\mathbb{Q}$. Ce programme ne se termine pas pour tous les rationnels. Par exemple, si $x = \frac{10}{3}$, le programme ne se termine pas, donc il n'y aura pas de fonction de rang dans le domaine des rationnels.

En considérant seulement le domaine des entiers relatifs, $\mathbb{Z}$, ce programme se termine. La fonction f non linéaire suivante le prouve :

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 2 & 0 < x \leq 2 \\ 4 & x = 3 \\ 3 & x = 4 \\ 1 & x \geq 5 \end{cases}$$

Pour vérifier qu'une fonction de rang corresponde bien à un programme, il faut démontrer que la fonction est bien strictement décroissante ($f(x^0) \geq 1 + f(x^1)$) et positive ($f(x^0) \geq 0$) pour chaque état du programme.

Pour le cas de la fonction f :

x^0	x^1	$f(x^0) \geq 1 + f(x^1)$	$f(x^0) \geq 0$
1	8	$2 \geq 1 + 1$	$2 \geq 0$
2	6	$2 \geq 1 + 1$	$2 \geq 0$
3	4	$4 \geq 1 + 3$	$4 \geq 0$
4	2	$3 \geq 1 + 2$	$3 \geq 0$
5	0	$1 \geq 1 + 0$	$1 \geq 0$
6	-2	$1 \geq 1 + 0$	$1 \geq 0$
⋮	⋮	⋮	⋮

Comment pourrions-nous démontrer que cet exemple se termine avec la méthode de Chen ?
Une possibilité serait de remplacer l'utilisation de la méthode de Podelski & Rybalchenko par une méthode capable de trouver une fonction de rang dans les entiers relatifs uniquement. Cette piste n'a pas été testée, son résultat n'est donc pas garanti.

4.2 Deuxième exemple

Comme programme linéaire se terminant, nous prenons pour exemple la boucle $n^o\,3$ de la suite de tests de Chen [6].

```
int x
while (x > 1)
        −2x := x
```

$$L \triangleq \langle \{x^0 > 1\}, \{-2x^1 = x^0\}, 0, 1 \rangle$$

La boucle L est linéaire, on peut donc directement appliquer la méthode de Podelski & Rybalchenko pour trouver une fonction de rang linéaire :

1. La première étape consiste à transformer la boucle $L_{1.2}$ en système d'inéquations :

$$\begin{cases} x^0 & > & 1 \\ -2x^1 & \leq & x^0 \\ -2x^1 & \geq & x^0 \end{cases} \Leftrightarrow \begin{cases} -x^0 & \leq & -1 \\ -x^0 - 2x^1 & \leq & 0 \\ x^0 + 2x^1 & \leq & 0 \end{cases}$$

2. Ensuite, nous écrivons le système sous forme matricielle $(A^0 A^1)X \leq b$:

$$\begin{pmatrix} -1 & 0 \\ -1 & -2 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x^0 \\ x^1 \end{pmatrix} \leq \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}$$

3. Nous pouvons dès lors résoudre le système d'inéquations en posant $\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3)$ et $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3)$:

$$\begin{cases} \lambda_1, \lambda_2 & \geq & 0 \\ \lambda_1 A^1 & = & 0 \\ (\lambda_1 - \lambda_2)A^0 & = & 0 \\ \lambda_2(A^0 + A^1) & = & 0 \\ \lambda_2 b & < & 0 \end{cases} \rightarrow \begin{cases} \lambda_1, \lambda_2 \geq 0 \\ -2\lambda'_2 + 2\lambda'_3 = 0 \\ -\lambda'_1 + \lambda''_1 - \lambda'_2 + \lambda''_2 + \lambda'_3 - \lambda''_3 = 0 \\ -\lambda''_1 - 3\lambda''_2 + \lambda''_3 = 0 \\ -\lambda''_1 < 0 \end{cases}$$

Grâce à SWI-Prolog, nous avons trouvé une solution à ce système. En effet, les valeurs suivantes permettent la résolution du système :

$$\begin{cases} \lambda'_2, \lambda'_3, \lambda''_2 & = & 0 \\ \lambda'_1 & = & \frac{2}{3} \\ \lambda''_1 & = & 1 \\ \lambda''_3 & = & \frac{1}{3} \end{cases} \qquad \lambda_1 = (\frac{2}{3}, 0, 0), \lambda_2 = (1, 0, \frac{1}{3})$$

4. Enfin, nous pouvons trouver la fonction de rang $\rho_1(X^0)$ à l'aide de la formule suivante :

$$\rho(X^0) = \begin{cases} rX^0 & \text{s'il } \exists X^1 \text{ tel que } (A^0 A^1) \begin{pmatrix} x^0 \\ x^1 \end{pmatrix} \leq b \\ \delta_0 - \delta & \text{si non} \end{cases}$$

où $r \triangleq \lambda_2 A^1, \delta_0 \triangleq -\lambda_1 b$, et $\delta \triangleq -\lambda_2 b$

5. Voici donc notre fonction de rang trouvée :

$$\rho_1(X^0) = \begin{cases} \frac{2}{3}x^0 & \text{s'il } \exists X^1 \text{ tel que } (A^0 A^1)\begin{pmatrix} X^0 \\ X^1 \end{pmatrix} \leq b \\ -\frac{1}{3} & \text{si non} \end{cases}$$

La méthode de Podelski & Rybalchenko a trouvé une fonction de rang. Le programme se termine.

Conclusion : Ce programme est linéaire. La méthode de Podeslki & Rybalchenko trouve donc directement une fonction de rang. Étant donné que nous avons trouvé une fonction de rang correspondant à notre programme, nous pouvons conclure qu'il se termine.
La fonction de rang de ce programme est :

$$f(x) = \begin{cases} \frac{2}{3}x & x > 1 \\ -\frac{1}{3} & x \leq 1 \end{cases}$$

4.3 Troisième exemple

Comme programme non linéaire se terminant, nous prenons pour exemple la boucle n^o 21 de la suite de tests de Chen [6].

```
int x, y
while (x > 0)
        x := y
        y := y − 1
```

$$L \triangleq \langle \{x^0 > 0\}, \{x^1 = y^0, y^1 = y^0 - 1\}, 0, 1 \rangle$$

Nous constatons que L n'est pas linéaire, donc nous séparons notre boucle LSL en suivant la méthode *diviser pour régner*.
Comme la pratique courante le veut, nous utilisons COND_L comme SEED. Nous obtenons alors :

$$\begin{array}{lcll} L_{1.1} & = & \langle \text{COND}_L, \text{UPDATE}_L \cup \{x^0 > x^1\}, 0, 1 \rangle & \text{(cas trivial)} \\ L_{1.2} & = & \langle \text{COND}_L, \text{UPDATE}_L \cup \{x^0 \leq x^1\}, 0, 1 \rangle & \text{(cas synthétisé)} \end{array}$$

Cas trivial : x^1 est inférieur à x^0 et on a une borne inférieure, on a donc une fonction de rang linéaire : $\rho_1(X^0) = x^0$

Cas synthétisé : on teste par la méthode de Podelski & Rybalchenko si on a une fonction de rang linéaire :

1. La première étape consiste à transformer la boucle $L_{1.2}$ en système d'inéquations :

$$\begin{cases} x^0 > 0 \\ x^1 \leq y^0 \\ x^1 \geq y^0 \\ y^1 \leq y^0 - 1 \\ y^1 \geq y^0 - 1 \\ x^0 \leq x^1 \end{cases} \Leftrightarrow \begin{cases} -x^0 \leq 0 \\ -y^0 + x^1 \leq 0 \\ y^0 - x^1 \leq 0 \\ -y^0 + y^1 \leq -1 \\ y^0 - y^1 \leq 1 \\ x^0 - x^1 \leq 0 \end{cases}$$

2. Ensuite, écrivons le système sous forme matricielle $(A^0 A^1)X \leq b$:

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x^0 \\ y^0 \\ x^1 \\ y^1 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 1 \\ 0 \end{pmatrix}$$

3. Nous pouvons dès lors résoudre le système d'inéquations en posant
$\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3, \lambda'_4, \lambda'_5, \lambda'_6)$ et $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3, \lambda''_4, \lambda''_5, \lambda''_6)$:

$$\begin{cases} \lambda_1, \lambda_2 \geq 0 \\ \lambda_1 A^1 = 0 \\ (\lambda_1 - \lambda_2)A^0 = 0 \\ \lambda_2(A^0 + A^1) = 0 \\ \lambda_2 b < 0 \end{cases} \rightarrow \begin{cases} \lambda_1 \geq 0, \lambda_2 \geq 0 \\ \lambda'_2 - \lambda'_3 - \lambda'_6 = 0 \\ \lambda'_4 - \lambda'_5 = 0 \\ -\lambda'_1 + \lambda''_1 + \lambda'_6 - \lambda''_6 = 0 \\ -\lambda'_2 + \lambda''_2 + \lambda'_3 - \lambda''_3 - \lambda'_4 \\ \quad +\lambda''_4 + \lambda'_5 - \lambda''_5 = 0 \\ -\lambda''_1 + \lambda''_2 - \lambda''_3 = 0 \\ -\lambda''_2 + \lambda''_3 = 0 \\ -\lambda''_4 + \lambda''_5 < 0 \end{cases}$$

Grâce à SWI-Prolog, nous avons trouvé une solution à ce système. En effet, les valeurs suivantes permettent la résolution du système :

$$\lambda_1 = (1, 1, 0, 0, 0, 1), \lambda_2 = (0, 0, 0, 1, 0, 0)$$

4. Enfin, nous pouvons trouver la fonction de rang $\rho_2(X^0)$ à l'aide de la formule suivante :

$$\rho(X^0) = \begin{cases} rX^0 & \text{s'il } \exists X^1 \text{ tel que } (A^0 A^1) \begin{pmatrix} x^0 \\ y^0 \\ x^1 \\ y^1 \end{pmatrix} \leq b \\ \delta_0 - \delta & \text{si non} \end{cases}$$

où $r \triangleq \lambda_2 A^1, \delta_0 \triangleq -\lambda_1 b$, et $\delta \triangleq -\lambda_2 b$

5. Voici donc notre fonction de rang trouvée :

$$\rho_2(X^0) = \begin{cases} y^0 & \text{s'il } \exists X^1 \text{ tel que } (A^0 A^1) \begin{pmatrix} x^0 \\ y^0 \\ x^1 \\ y^1 \end{pmatrix} \leq b \\ 1 & \text{si non} \end{cases}$$

Étant donné que nous avons trouvé plus d'une fonction de rang linéaire, nous devons vérifier que l'union de ces relations de rang est bien incluse dans la fermeture transitive de la relation de transition. Pour cela, nous devons donc vérifier la relation $R^+ \subseteq T$, avec $T = \tau(\rho_1) \cup \tau(\rho_2)$. Nous pouvons réécrire cette relation sous la forme :

$$(R \subseteq T) \wedge ((R \circ T \subseteq T) \vee (T \circ R \subseteq T))$$

Pour faciliter la compréhension de la vérification, voici la valeur de quelques variables :

$$\begin{array}{lcl} R & = & \{(x^0, y^0, x^1, y^1) \mid x^0 > 0 \wedge x^1 = y^0 \wedge y^1 = y^0 - 1\} \\ T & = & \tau_1 \cup \tau_2 \\ \tau_1 & = & \{(x^0, y^0, x^1, y^1) \mid x^0 \geq 0 \wedge x^0 \geq x^1 + 1\} \\ \tau_2 & = & \{(x^0, y^0, x^1, y^1) \mid y^0 \geq 0 \wedge y^0 \geq y^1 + 1\} \end{array}$$

Passons à la vérification :

1. Pour commencer, nous allons vérifier $R \subseteq T$:

$$\begin{array}{rcl} R \subseteq T & = & \neg[\exists x^0, y^0, x^1, y^1 : R \wedge \neg\tau_1 \wedge \neg\tau_2] \\ & = & \neg[\exists x^0, y^0, x^1, y^1 : \\ & & (x^0 > 0 \wedge x^1 = y^0 \wedge y^1 = y^0 - 1) \\ & & \quad \wedge\neg(x^0 \geq 0 \wedge x^0 \geq x^1 + 1) \\ & & \quad \wedge\neg(y^0 \geq 0 \wedge y^0 \geq y^1 + 1)] \end{array}$$

$\exists x^0, y^0, x^1, y^1 : (x^0 > 0 \wedge x^1 = y^0 \wedge y^1 = y^0 - 1) \wedge \neg(x^0 \geq 0 \wedge x^0 \geq x^1 + 1) \wedge \neg(y^0 \geq 0 \wedge y^0 \geq y^1 + 1)$ est insatisfaisable selon le solveur Z3. $R \subseteq T$ est donc vérifiée.

2. Ensuite, nous allons vérifier $((R \circ T \subseteq T) \vee (T \circ R \subseteq T))$:
Nous choisissons arbitrairement de commencer par la vérification de $R \circ T \subseteq T$.

$$
\begin{array}{rcl}
R \circ T \subseteq T & = & (R \circ \tau_1) \cup (R \circ \tau_2) \Rightarrow \tau_1 \cup \tau_2 \\
& = & \neg[\exists(t^0, r^0, t^1, r^1) : (R \circ \tau_1) \cup (R \circ \tau_2) \wedge \neg\tau_1 \wedge \neg\tau_2] \\
R \circ \tau_1 & = & \{(t^0, t^1, r^{0'}, r^{1'}) \mid \exists t^{0'}, t^{1'}, r^0, r^1 : \\
& & \quad t^0 \geq 0 \wedge t^0 \geq t^1 + 1 \\
& & \quad \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \\
& & \quad \wedge r^0 > 0 \wedge r^{0'} = r^1 \wedge r^{1'} = r^1 - 1\} \\
R \circ \tau_2 & = & \{(t^0, t^1, r^{0'}, r^{1'}) \mid \exists t^{0'}, t^{1'}, r^0, r^1 : \\
& & \quad r^{0'} \geq 0 \wedge r^{0'} \geq r^{1'} + 1 \\
& & \quad \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \\
& & \quad \wedge r^0 > 0 \wedge r^{0'} = r^1 \wedge r^{1'} = r^1 - 1\} \\
R \circ T \subseteq T & = & \neg[\exists t^0, t^1, t^{0'}, t^{1'}, r^0, r^1, r^{0'}, r^{1'} : \\
& & \quad ((t^0 \geq 0 \wedge t^0 \geq t^1 + 1 \\
& & \qquad \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \\
& & \qquad \wedge r^0 > 0 \wedge r^{0'} = r^1 \wedge r^{1'} = r^1 - 1) \\
& & \quad \vee (r^{0'} \geq 0 \wedge r^{0'} \geq r^{1'} + 1 \\
& & \qquad \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \\
& & \qquad \wedge r^0 > 0 \wedge r^{0'} = r^1 \\
& & \qquad \wedge r^{1'} = r^1 - 1)) \\
& & \quad \wedge \neg(t^0 \geq 0 \wedge t^0 \geq t^{0'} + 1) \\
& & \quad \wedge \neg(t^1 \geq 0 \wedge t^1 \geq t^{1'} + 1)]
\end{array}
$$

Le solveur Z3 ne sait pas déterminer si la formule est satisfaisable.

$R \circ T \subseteq T$ n'est donc pas vérifiée. Nous allons donc vérifier $T \circ R \subseteq T$.

$$
\begin{aligned}
T \circ R \subseteq T &= (\tau_1 \circ R) \cup (\tau_2 \circ R) \Rightarrow \tau_1 \cup \tau_2 \\
&= \neg[\exists(t^0, r^0, t^1, r^1) : (\tau_1 \circ R) \cup (\tau_2 \circ R) \Rightarrow \neg\tau_1 \wedge \neg\tau_2] \\
\tau_1 \circ R &= \{(r^0, r^1, t^{0'}, t^{1'}) \mid \exists t^0, t^1, r^{0'}, r^{1'} : \\
&\qquad r^0 > 0 \wedge r^{0'} = r^1 \wedge r^{1'} = r^1 - 1 \\
&\qquad \wedge t^0 = r^{0'} \wedge t^1 = r^{1'} \\
&\qquad \wedge t^0 \geq 0 \wedge t^0 \geq t^1 + 1\} \\
\tau_2 \circ R &= \{(r^0, r^1, t^{0'}, t^{1'}) \mid \exists t^0, t^1, r^{0'}, r^{1'} : \\
&\qquad r^0 > 0 \wedge r^{0'} = r^1 \wedge r^{1'} = r^1 - 1 \\
&\qquad \wedge t^0 = r^{0'} \wedge t^1 = r^{1'} \\
&\qquad \wedge t^{0'} \geq 0 \wedge t^{0'} \geq t^{1'} + 1\} \\
T \circ R \subseteq T &= \neg[\exists r^0, r^1, r^{0'}, r^{1'}, t^0, t^1, t^{0'}, t^{1'} : \\
&\qquad ((r^0 > 0 \wedge r^{0'} = r^1 \wedge r^{1'} = r^1 - 1 \\
&\qquad\quad \wedge t^0 = r^{0'} \wedge t^1 = r^{1'} \\
&\qquad\quad \wedge t^0 \geq 0 \wedge t^0 \geq t^1 + 1) \\
&\qquad\ \vee (r^0 > 0 \wedge r^{0'} = r^1 \wedge r^{1'} = r^1 - 1 \\
&\qquad\quad \wedge t^0 = r^{0'} \wedge t^1 = r^{1'} \\
&\qquad\quad \wedge t^{0'} \geq 0 \wedge t^{0'} \geq t^{1'} + 1)) \\
&\qquad \wedge \neg(r^0 \geq 0 \wedge r^0 \geq r^1 + 1) \\
&\qquad \wedge \neg(t^0 \geq 0 \wedge t^0 \geq t^1 + 1)]
\end{aligned}
$$

Selon le solveur Z3 la formule est insatisfaisable. $T \circ R \subseteq T$ est donc vérifiée.

Étant donné que $R \subseteq T$ et $T \circ R \subseteq T$ sont vérifiées, $R^+ \subseteq T$ est vérifiée.

Conclusion : La méthode de Chen est capable de prouver que ce programme se termine. En effet, une disjonction de fonction de rang non linéaire $(f_1 \cup f_2)$ a été trouvée :

$$[f_1(x, y) = x] \cup \left[f_2(x, y) = \begin{cases} y & x > 0 \\ 1 & x \leq 0 \end{cases} \right]$$

De plus, la relation $R^+ \subseteq T$ est vérifiée et de ce fait il est prouvé que le programme se termine.

4.4 Quatrième exemple

Comme programme non linéaire se terminant, nous prenons l'exemple 3.1 utilisé dans l'article « Eventual Linear Ranking Functions » [2].

int x, y
while $(x \geq 0)$
$\quad y \leq y - 1$
$\quad x \leq x + y$

$$L \triangleq \langle \{x^0 \geq 0\}, \{y^1 \leq y^0 - 1, x^1 \leq x^0 + y^0\}, 0, 1 \rangle$$

À première vue, nous ne sommes pas directement capable de constater si L est linéaire. Nous allons donc appliquer la méthode de Podelski & Rybalchenko pour répondre à cette question. Cette méthode étant complète, si une fonction de rang existe, la méthode nous l'indiquera. Commençons donc par appliquer la méthode de Podelski & Rybalchenko.

1. La première étape consiste à transformer la boucle L en système d'inéquations :

$$\begin{cases} x^0 & \geq & 0 \\ y^1 & \leq & y^0 - 1 \\ x^1 & \leq & x^0 + y^0 \end{cases} \Leftrightarrow \begin{cases} -x^0 & \leq & 0 \\ -y^0 + y^1 & \leq & -1 \\ -x^0 - y^0 + x^1 & \leq & 0 \end{cases}$$

2. Ensuite, nous écrivons le système sous forme matricielle $(A^0 A^1)X \leq b$:

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x^0 \\ y^0 \\ x^1 \\ y^1 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}$$

3. Nous pouvons dès lors résoudre le système d'inéquations en posant $\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3)$ et $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3)$:

$$\begin{cases} \lambda_1, \lambda_2 & \geq & 0 \\ \lambda_1 A^1 & = & 0 \\ (\lambda_1 - \lambda_2) A^0 & = & 0 \\ \lambda_2 (A^0 + A^1) & = & 0 \\ \lambda_2 b & < & 0 \end{cases} \rightarrow \begin{cases} \lambda_1 \geq 0, \lambda_2 \geq 0 \\ \lambda'_3 = 0 \\ \lambda'_2 = 0 \\ -\lambda'_1 + \lambda''_1 - \lambda'_3 + \lambda''_3 = 0 \\ -\lambda'_2 + \lambda''_2 - \lambda'_3 + \lambda''_3 = 0 \\ -\lambda''_1 = 0 \\ -\lambda''_3 = 0 \\ -\lambda''_2 < 0 \end{cases}$$

Aucune solution au système d'inéquations n'a été trouvée à l'aide de SWI-Prolog. Nous en déduisons donc que la boucle L n'est pas linéaire. Il faut donc séparer notre boucle LSL en suivant la méthode *diviser pour régner*. Comme la pratique courante le veut, nous utilisons $\textsc{Cond}_L$ comme SEED. Nous obtenons alors :

$$\begin{array}{lcll} L_{1.1} & = & \langle \textsc{Cond}_L, \textsc{Update}_L \cup \{x^0 > x^1\}, 0, 1 \rangle & \text{(cas trivial)} \\ L_{1.2} & = & \langle \textsc{Cond}_L, \textsc{Update}_L \cup \{x^0 \leq x^1\}, 0, 1 \rangle & \text{(cas synthétisé)} \end{array}$$

Analysons les différents cas :

Cas trivial : x^1 est inférieur à x^0 et on a une borne inférieure, on a donc une fonction de rang linéaire : $\rho_1(X^0) = x^0$

Cas synthétisé : on teste par la méthode de Podelski & Rybalchenko si on a une fonction de rang linéaire :

1. La première étape consiste à transformer la boucle $L_{1.2}$ en système d'inéquations :

$$\begin{cases} x^0 \geq 0 \\ y^1 \leq y^0 - 1 \\ x^1 \leq x^0 + y^0 \\ x^0 \leq x^1 \end{cases} \Leftrightarrow \begin{cases} -x^0 \leq 0 \\ -y^0 + y^1 \leq -1 \\ -x^0 - y^0 + x^1 \leq 0 \\ x^0 - x^1 \leq 0 \end{cases}$$

2. Ensuite, écrivons le système sous forme matricielle $(A^0 A^1)X \leq b$:

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \\ 1 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x^0 \\ y^0 \\ x^1 \\ y^1 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \end{pmatrix}$$

3. Nous pouvons dès lors résoudre le système d'inéquations en posant $\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3, \lambda'_4)$ et $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3, \lambda''_4)$:

$$\begin{cases} \lambda_1, \lambda_2 \geq 0 \\ \lambda_1 A^1 = 0 \\ (\lambda_1 - \lambda_2)A^0 = 0 \\ \lambda_2(A^0 + A^1) = 0 \\ \lambda_2 b < 0 \end{cases} \rightarrow \begin{cases} \lambda_1 \geq 0, \lambda_2 \geq 0 \\ \lambda'_3 - \lambda'_4 = 0 \\ -\lambda'_1 + \lambda'_2 = 0 \\ -\lambda'_1 + \lambda''_1 - \lambda'_3 + \lambda''_3 + \lambda'_4 - \lambda''_4 = 0 \\ -\lambda'_2 + \lambda''_2 - \lambda'_3 + \lambda''_3 = 0 \\ -\lambda''_1 = 0 \\ -\lambda''_1 - \lambda''_3 = 0 \\ -\lambda''_2 < 0 \end{cases}$$

Grâce à SWI-Prolog, nous avons trouvé une solution à ce système. En effet, les valeurs suivantes permettent la résolution du système :

$$\lambda_1 = (0, 0, 1, 1), \lambda_2 = (0, 1, 0, 0)$$

4. Enfin, nous pouvons trouver la fonction de rang $\rho_1(X^0)$ à l'aide de la formule suivante :

$$\rho(X^0) = \begin{cases} rX^0 & \text{s'il } \exists X^1 \text{ tel que } (A^0 A^1) \begin{pmatrix} x^0 \\ y^0 \\ x^1 \\ y^1 \end{pmatrix} \leq b \\ \delta_0 - \delta & \text{si non} \end{cases}$$

où $r \triangleq \lambda_2 A^1, \delta_0 \triangleq -\lambda_1 b$, et $\delta \triangleq -\lambda_2 b$

5. Voici donc notre fonction de rang trouvée :

$$\rho_2(X^0) = \begin{cases} y^0 & \text{s'il } \exists X^1 \text{ tel que } (A^0 A^1) \begin{pmatrix} x^0 \\ y^0 \\ x^1 \\ y^1 \end{pmatrix} \leq b \\ -1 & \text{si non} \end{cases}$$

Étant donné que nous avons trouvé plus d'une fonction de rang linéaire, nous devons vérifier que l'union de ces relations de rang est bien incluse dans la fermeture transitive de la relation de transition. Pour cela, nous devons donc vérifier la relation $R^+ \subseteq T$, avec $T = \tau(\rho_1) \cup \tau(\rho_2)$. Nous pouvons réécrire cette relation sous la forme :

$$(R \subseteq T) \wedge ((R \circ T \subseteq T) \vee (T \circ R \subseteq T))$$

Pour faciliter la compréhension de la vérification, voici la valeur de quelques variables :

$$\begin{array}{rcl} R & = & \{(x^0, y^0, x^1, y^1) \mid x^0 \geq 0 \wedge y^1 \leq y^0 - 1 \wedge x^1 \leq x^0 + y^0\} \\ T & = & \tau_1 \cup \tau_2 \\ \tau_1 & = & \{(x^0, y^0, x^1, y^1) \mid x^0 \geq 0 \wedge x^0 \geq x^1 + 1\} \\ \tau_2 & = & \{(x^0, y^0, x^1, y^1) \mid y^0 \geq 0 \wedge y^0 \geq y^1 + 1\} \end{array}$$

Passons à la vérification :

1. Pour commencer, nous allons vérifier $R \subseteq T$:

$$\begin{array}{rcl} R \subseteq T & = & \neg[\exists x^0, y^0, x^1, y^1 : R \wedge \neg\tau_1 \wedge \neg\tau_2] \\ & = & \neg[\exists x^0, y^0, x^1, y^1 : \\ & & (x^0 \geq 0 \wedge y^1 \leq y^0 - 1 \wedge x^1 \leq x^0 + y^0) \\ & & \quad \wedge\neg(x^0 \geq 0 \wedge x^0 \geq x^1 + 1) \\ & & \quad \wedge\neg(y^0 \geq 0 \wedge y^0 \geq y^1 + 1)] \end{array}$$

$\exists x^0, y^0, x^1, y^1 : (x^0 \geq 0 \wedge y^1 \leq y^0 - 1 \wedge x^1 \leq x^0 + y^0) \wedge \neg(x^0 \geq 0 \wedge x^0 \geq x^1 + 1) \wedge \neg(y^0 \geq 0 \wedge y^0 \geq y^1 + 1)$ est insatisfaisable selon le solveur Z3. $R \subseteq T$ est donc vérifiée.

2. Ensuite, nous allons vérifier $((R \circ T \subseteq T) \vee (T \circ R \subseteq T))$:

$$
\begin{array}{rcl}
R \circ T \subseteq T & = & (R \circ \tau_1) \cup (R \circ \tau_2) \Rightarrow \tau_1 \cup \tau_2 \\
& = & \neg[\exists(t^0, r^0, t^1, r^1) : (R \circ \tau_1) \cup (R \circ \tau_2) \wedge \neg\tau_1 \wedge \neg\tau_2] \\
R \circ \tau_1 & = & \{(t^0, t^1, r^{0'}, r^{1'}) \mid \exists t^{0'}, t^{1'}, r^0, r^1 : \\
& & \quad t^0 \geq 0 \wedge t^0 \geq t^{0'} + 1 \\
& & \quad \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \\
& & \quad \wedge r^0 > 0 \wedge r^{1'} \leq r^1 - 1 \wedge r^{0'} \leq r^0 + r^1\} \\
R \circ \tau_2 & = & \{(t^0, t^1, r^{0'}, r^{1'}) \mid \exists t^{0'}, t^{1'}, r^0, r^1 : \\
& & \quad t^1 \geq 0 \wedge t^1 \geq t^{1'} + 1 \\
& & \quad \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \\
& & \quad \wedge r^0 > 0 \wedge r^{1'} \leq r^1 - 1 \wedge r^{0'} \leq r^0 + r^1\} \\
R \circ T \subseteq T & = & \neg[\exists t^0, t^1, t^{0'}, t^{1'}, r^0, r^1, r^{0'}, r^{1'} : \\
& & \quad ((t^0 \geq 0 \wedge t^0 \geq t^{0'} + 1 \\
& & \quad\quad \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \\
& & \quad\quad \wedge r^0 > 0 \wedge r^{1'} \leq r^1 - 1 \wedge r^{0'} \leq r^0 + r^1) \\
& & \quad \vee \\
& & \quad (t^1 \geq 0 \wedge t^1 \geq t^{1'} + 1 \\
& & \quad\quad \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \\
& & \quad\quad \wedge r^0 > 0 \wedge r^{1'} \leq r^1 - 1 \wedge r^{0'} \leq r^0 + r^1)) \\
& & \quad \wedge \neg(t^0 \geq 0 \wedge t^0 \geq t^{0'} + 1) \\
& & \quad \wedge \neg(t^1 \geq 0 \wedge t^1 \geq t^{1'} + 1)]
\end{array}
$$

$\exists t^0, t^1, t^{0'}, t^{1'}, r^0, r^1, r^{0'}, r^{1'} : ((t^0 \geq 0 \wedge t^0 \geq t^{0'} + 1 \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \wedge r^0 > 0 \wedge r^{1'} \leq r^1 - 1 \wedge r^{0'} \leq r^0 + r^1) \vee (t^1 \geq 0 \wedge t^1 \geq t^{1'} + 1 \wedge t^{0'} = r^0 \wedge t^{1'} = r^1 \wedge r^0 > 0 \wedge r^1 \leq r^1 - 1 \wedge r^{0'} \leq r^0 + r^1)) \wedge \neg(t^0 \geq 0 \wedge t^0 \geq t^{0'} + 1) \wedge \neg(t^1 \geq 0 \wedge t^1 \geq t^{1'} + 1)$ est insatisfaisable selon le solveur Z3. $R \circ T \subseteq T$ est donc vérifiée.

Étant donné que $R \subseteq T$ et $R \circ T \subseteq T$ sont vérifiées, $R^+ \subseteq T$ est vérifiée.

Conclusion : La méthode de Chen est capable de prouver que ce programme se termine. En effet, une disjonction de fonction de rang non linéaire $(f_1 \cup f_2)$ a été trouvée :

$$[f_1(x, y) = x] \cup \left[f_2(x, y) = \left\{ \begin{array}{ll} y & x \geq 0 \\ -1 & x < 0 \end{array} \right. \right]$$

De plus, la relation $R^+ \subseteq T$ est vérifiée et de ce fait il est prouvé que le programme se termine.

Chapitre 5

Récents démonstrateurs

Les trois articles résumés dans ce chapitre portent sur des récents démonstrateurs de terminaison qui se basent sur des arguments de terminaison disjonctifs.
Ces résumés vont permettre de comparer la méthode de Chen étudiée en détail et d'autres développements récents.
Deux articles (section 5.1 [9] et section 5.3 [5]) implémentent une stratégie basée sur le raffinement de l'argument de terminaison. Quant au troisième (section 5.2 [3]), il implémente une stratégie basée sur l'analyse variante. Cependant, ces trois articles tournent autour du même fait : « un programme termine si et seulement si sa fermeture transitive est incluse dans une disjonction finie de relation bien fondée ».

5.1 Démonstrateur de terminaison de programme : Terminator

Cette section présente TERMINATOR, un démonstrateur de terminaison de programme. Il est capable de supporter de grandes parties de code (plus de 20 000 lignes de code) et des langages de programmation avec des capacités avancées telles que les boucles imbriquées, les pointeurs, les effets de bord, etc. Ce démonstrateur a été développé par Byron Cook, Andreas Podelski et Andrey Rybalchenko en 2006 [9].

Tout d'abord, une description générale de TERMINATOR sera donnée à la section 5.1.1. L'algorithme formel de ce démonstrateur suivra à la section 5.1.2. Pour finir, un exemple d'utilisation de cet algorithme sera présenté à la section 5.1.4.

5.1.1 Description de Terminator

L'outil TERMINATOR se distingue des autres méthodes de preuve de terminaison de programme, par la manière dont il passe entre les tâches de

construction et de vérification des arguments de terminaison. En comparaison aux méthodes existantes, TERMINATOR construit facilement ses arguments de terminaison, mais requiert plus d'efforts pour les vérifier. Ceci s'explique par les deux raisons suivantes :

1. On ne construit pas un seul argument de terminaison, mais un ensemble d'arguments qui est potentiellement valide. Tous les arguments ne doivent pas être valides. On obtient donc un sur-ensemble de fonctions de rang.
2. Du fait que l'argument de terminaison est un ensemble et non une simple fonction de rang, sa vérification est plus compliquée.
   Pour rappel, vérifier la validité d'une seule fonction de rang consiste à montrer qu'après avoir exécuté une simple transition, le rang diminue. Pour vérifier un ensemble de fonctions de rang, il faut considérer toutes les séquences finies de transitions. Il faut montrer que pour chaque séquence, une fonction de rang diminue avant et après un état. Formellement, il faut d'abord trouver toutes les paires d'états s_1 et s_2 tel que s_1 est atteignable à partir de l'état initial du programme et s_2 est atteignable à partir de s_1. Ensuite, on montre que la valeur d'une des fonctions de rang diminue de s_1 à s_2. On appelle cela une analyse d'accessibilité binaire. Cette analyse sera détaillée à la section 5.1.3

5.1.2 Terminator

Listing 5.1 Algorithme de TERMINATOR

```
1  input
2          R : relation de transition d'un programme P
3          I : ensemble d'états initiaux
4  output
5          "terminant"      si P se termine
6          "non terminant"  si non
7  begin
8          T := ∅
9          repeat
10                 if R_I^+ ⊆ T then
11                         return "terminant"
12                 else
13                         ρ := relation binaire tel que ρ ⊆ R_I^+, mais ρ ⊈ T
14
15                 if ρ n'est pas une relation bien fondée then
16                         return "non terminant"
17                 else
18                         W := relation de rang
19                         T := T ∪ W
20         end
21 end.
```

L'algorithme TERMINATOR 5.1 consiste en deux parties :

1. La première partie consiste à vérifier l'accessibilité binaire de T (lignes 10 à 13) :
   - T est l'argument de terminaison, donc l'union des relations bien fondées ;
   - I est l'ensemble des états initiaux ;
   - R_I^+ est la fermeture transitive de R restreint aux états atteignables, à partir des états initiaux I.
     Ce sont donc les paires d'états (s_1, s_2) tels que s_2 est atteignable à partir de s_1 en une étape et s_1 est atteignable à partir d'un état initial s_0. Formellement,
     $$R_I^+ = \{(s_1, s_2) \mid \exists s_0 \in I.(s_0, s_1) \in R^* \wedge (s_1, s_2) \in R^+\}$$
   - L'analyse d'accessibilité binaire est une procédure qui vérifie quand R_I^+ est contenue dans une relation binaire T :
     $$R_I^+ \subseteq T$$
   - Dans le cas où l'analyse d'accessibilité binaire échoue, c'est-à-dire que $R_I^+ \not\subseteq T$, il existe une séquence non-vide $\tau_1, \ldots, \tau_i, \ldots, \tau_n$ avec une séquence d'exécution $s_0 \rightarrow_{\tau_1} s_1 \ldots s_{i-1} \rightarrow_{\tau_i} s_i \ldots s_{n-1} \rightarrow_{\tau_n} s_n$ telle que la paire d'états (s_i, s_n) n'est pas dans T, formellement $(s_i, s_n) \in R_I^+$ mais $(s_i, s_n) \notin T$.
   - Soit ρ la relation consistant en toutes les paires d'états (s_i, s_n) qui sont connectées par une séquence d'exécution décrite au point précédent.
   - La relation ρ est donc le contre-exemple de l'inclusion $R_I^+ \subseteq T$. Nous avons donc $\rho \subseteq R_I^+$ et $\rho \not\subseteq T$.
2. La deuxième partie consiste à chercher une fonction de rang pour ρ (lignes 15 à 19) :
   Pour trouver la fonction de rang de ρ, on applique la méthode complète pour prouver la terminaison de boucles linéaires simples développée par A. Podelski et A. Rybalchenko [15]. Si une fonction de rang existe pour le programme, la méthode la trouvera. Cette méthode a été décrite dans ce mémoire à la section 2.1.

5.1.3 Analyse d'accessibilité binaire

L'analyse d'accessibilité binaire est la procédure qui vérifie que R_I^+ est contenu dans une relation binaire T.
Formellement,
$$R_I^+ \subseteq T$$

Cette analyse requiert différentes étapes :

1. Caractériser R_I^+ par le plus petit point fixe d'une fonction F.
2. Transformer P, sur base de la structure de F. Cette transformation crée un nouveau programme $\widehat{P}$ qui implémente F. Ceci signifie que le plus petit point fixe de F est équivalent à l'ensemble des états atteignables dans $\widehat{P}$.
3. Décrire une transformation de $\widehat{P}$ qui crée $\widehat{P}_T$. Cette transformation a pour objectif d'arrêter le processus dès qu'il parait évident que l'inclusion n'est pas valide.
4. Transformer un chemin d'erreur de $\widehat{P}_T$ en une entrée pour le synthétiseur de fonction de rang.

Caractérisation par point fixe

Premièrement, on définit une fonction F dont le plus petit point fixe est R_I^+.
Le domaine de F consiste en des relations binaires sur des états d'un programme donné.
Le plus petit élément est la relation de transition retreinte aux états initiaux.

$$\bot \triangleq \{(s_1, s_2) \mid s_1 \in I \wedge (s_1, s_2) \in R\}$$

Avant de formaliser F, on définit une fonction auxiliaire `id` qui restreint la relation d'identité sur l'état du programme à l'image de sa relation d'entrée, formellement :

$$\texttt{id}(X) \triangleq \{(s_2, s_2) \mid \exists s_1.(s_1, s_2) \in X\}$$

Soit $\circ$, l'opérateur de composition relationnelle :

$$X \circ Y \triangleq \{(s_1, s_3) \mid \exists s_2.(s_1, s_2) \in X \wedge (s_2, s_3) \in Y\}$$

La fonction F prend une relation binaire X comme entrée.
Elle retourne, la composition relationnelle de l'union de X et la relation d'identité restreinte au deuxième composant, avec la relation de transition R du programme.

$$F(X) \triangleq (X \cup \texttt{id}(X)) \circ R$$

En effet, F :

— soit copie le composant de droite de X à gauche avant de le passer à R ;

— soit passe X tout simplement à R.

On peut maintenant définir que **la relation d'accessibilité binaire R_I^+ du programme** P est le plus petit point fixe de la fonction F sur le domaine des relations binaires, avec $\bot$ comme plus petit élement :

$$R_I^+ = \texttt{lfp}(F, \bot)$$

Transformation : $P \implies \widehat{P}$

On définit une transformation $\widehat{P}$ du programme P et une relation d'équivalence $\simeq$ entre des états de P et des états de $\widehat{P}$. $\widehat{P}$ sera donc un nouveau programme dérivé du programme P.
La transformation reflète la structure de la fonction F sur $\bot$ et l'ensemble des états atteignables de $\widehat{P}$.

— $V = \{v_1, \ldots, v_n, \texttt{pc}\}$ est l'ensemble des variables de P, incluant le compteur ordinal.
Le compteur ordinal ou en termes anglais *program counter*, est généralement abrégé `pc` et représente la variable qui contient l'adresse physique d'une instruction en cours d'exécution dans un processeur.

— $\widehat{V}$ est l'ensemble des variables de $\widehat{P}$ contenant V et un ensemble de variables dupliquées $'V = \{'v_1, \ldots, 'v_n, '\texttt{pc}\}$

— Un état $\widehat{s}$ de $\widehat{P}$ est équivalent à une paire d'états (s_1, s_2) de P. On note : $\widehat{s} \simeq (s_1, s_2)$, si les conditions suivantes sont vérifiées :

$$\begin{array}{rclcrcl} \widehat{s}('v_1) & = & s_1(v_1) & & \widehat{s}(v_1) & = & s_2(v_1) \\ & \cdots & & & & \cdots & \\ \widehat{s}('v_n) & = & s_1(v_n) & & \widehat{s}(v_n) & = & s_2(v_n) \\ \widehat{s}('\texttt{pc}) & = & s_1(\texttt{pc}) & & \widehat{s}(\texttt{pc}) & = & s_2(\texttt{pc}) \end{array}$$

Notez que $s_1(v_1)$ représente la valeur de la variable v_1 à l'état s_1 du programme P. Respectivement, $\widehat{s}('v_1)$ représente la valeur de la variable $'v_1$ à l'état $\widehat{s}$ du programme $\widehat{P}$ ainsi que $\widehat{s}(v_1)$ représente la valeur de la variable v_1 à l'état $\widehat{s}$ du programme $\widehat{P}$.

— $\widehat{I}$ est l'ensemble des états initiaux de $\widehat{P}$:

$$\widehat{I} \triangleq \{\widehat{s} \mid \exists s_0 \in I . \widehat{s} \simeq (s_0, s_0)\}$$

— Le plus petit point fixe de la fonction F sur le domaine des relations binaires, avec $\bot$ comme plus petit élément est équivalent à l'ensemble des états du programme transformé $\widehat{P}$ atteignable après au moins une étape. Formellement nous obtenons :

- $\texttt{lfp}(F, \bot)$ représentant le plus petit point fixe de la fonction F sur le domaine des relations binaires. $\bot$ étant le plus petit élément ;
- $\texttt{post}^+_{\widehat{P}}(\widehat{I})$ représentant l'ensemble des états du programme transformé $\widehat{P}$ atteignable après au moins une étape ;
- $\texttt{lfp}(F, \bot) = \texttt{post}^+_{\widehat{P}}(\widehat{I})$ représentant l'équivalance entre les deux points précédents.

Dès lors, nous pouvons implémenter l'analyse d'accessibilité binaire de la manière suivante :

1. Créer le programme transformé $\widehat{P}$
2. Générer un ensemble d'états atteignables $\texttt{post}^{+}_{\widehat{P}}(\widehat{I})$
3. Vérifier l'inclusion entre l'ensemble généré et T

Transformation : $\widehat{P} \implies \widehat{P}_T$

En pratique, il faut arrêter la génération de ce qui est atteignable dès qu'il est évident que l'inclusion n'est pas valide.
Pour cela, il faut effectuer une nouvelle transformation sur $\widehat{P}$.

Cette nouvelle transformation prend le programme $\widehat{P}$ et produit $\widehat{P}_T$ en remplaçant chaque déclaration composée de $\widehat{P}$ (sauf la déclaration initiale) :

```
L:      if (nondet()) { ... }
```

par les déclarations :

```
L:      if (! (T_L)) { ERROR: skip; }
        if (nondet()) { ... }
```

Notez que T_L représente l'argument de terminaison. Au commencement de la procédure, il est vide. Dans ce cas $T_L = \texttt{false}$.

Nous obtenons donc que l'inclusion $R_I^+ \subseteq T$ est valide si et seulement si l'erreur n'est pas atteignable dans le programme $\widehat{P}_T$.

Analyse du chemin d'erreur de $\widehat{P}_T$

On suppose qu'un vérificateur de sécurité temporelle peut produire un chemin d'erreur si l'erreur est atteignable.
On va interpréter un chemin d'erreur π :

1. Extraire un contre-exemple ρ à l'inclusion $R_I^+ \subseteq T$, provenant du chemin d'erreur τ.
   - τ traverse forcément à un moment la branche positive de la condition ajoutée par la transformation $P \implies \widehat{P}$;
   - on sépare τ à la dernière appartenance d'une telle déclaration en *tige* et *cycle*.
     tige représente les états avant d'arriver dans une boucle d'erreur et *cycle* représente la boucle d'erreur.
   - R_{tige} et R_{cycle} sont respectivement la relation de transition de tige et cycle ;
   -
   $$\rho \triangleq \{s_2, s_3) \mid \exists s_1 \in I.(s_1, s_2) \in R_{tige} \wedge (s_2, s_3) \in R_{cycle}\}$$

2. Si le synthétiseur de fonction de rang échoue pour la relation ρ, *tige* et *cycle* constituent potentiellement un contre-exemple à la terminaison du programme P.

5.1.4 Exemple d'application de Terminator

Pour illustrer cette méthode, nous allons l'appliquer sur le programme 5.2.

Listing 5.2 Programme P pour l'exemple d'application de TERMINATOR

```
1  int x := nondet(), y := nondet(), z := nondet()
2  if (y > 0)
3          do {
4                  if (nondet())
5                          x = x + y
6                  else
7                          z = x − y
8          } while (x < y ∧ y < z)
```

Première itération :

$T =$ `false`

1. Vérifier la validité de T à l'aide de l'analyse d'accessibilité binaire. L'analyse d'accessibilité binaire crée un nouveau programme 5.3 et y effectue l'analyse d'accessibilité.

   Listing 5.3 Exemple d'application de TERMINATOR, première itération $\widehat{P}$

   ```
   1          int x := nondet(), y := nondet(), z := nondet()
   2          if (y > 0)
   3                  do {
   3.1                        if ('pc == 5)
   3.2                                if (not (false))
   3.3                                        ERROR : skip
   3.4                        if ('pc == 0)
   3.5                                if (nondet())
   3.6                                        'x := x
   3.7                                        'y := y
   3.8                                        'z := z
   3.9                                        'pc := 5
   4                          if (nondet())
   5                                  x := x + y
   6                          else
   7                                  z := x − y
   9                  } while (x < y ∧ y < z)
   ```

2. Contre-exemple atteignant la déclaration `ERROR` :

$$1 \to 2 \to 3 \to 3.1 \to 3.4 \to 3.5 \to 3.6 \to 3.7 \to 3.8 \to 3.9$$

$$\to 4 \to 5 \to 6 \to 9 \to 3 \to 3.1 \to 3.2 \to 3.3$$

3. Ce contre-exemple peut être séparé en une *tige* et un *cycle* :

$$\begin{aligned} tige &= 1 \to 2 \to 3 \\ cycle &= 4 \to 5 \to 6 \to 9 \to 3 \end{aligned}$$

On obtient donc les relations suivantes :

$$\begin{aligned} R_{tige}((x^0,y^0,z^0),(x^1,y^1,z^1)) &\triangleq (y^1 > 0) \wedge (x^1 = x^0) \wedge (z^1 = z^0) \\ R_{cycle}((x^0,y^0,z^0),(x^1,y^1,z^1)) &\triangleq (x^1 = x^0 + y^0) \wedge (z^1 = z^0) \\ &\quad \wedge (y^1 = y^0) \wedge (x^1 < y^1) \\ &\quad \wedge (y^1 < z^1) \end{aligned}$$

4. On appelle un synthétiseur de fonction de rang, avec le contre-exemple ρ_1

$$\rho_1(s_1,s_2) \triangleq \exists s_0 \in I.(s_0,s_1) \in R_{tige} \wedge (s_1,s_2) \in R_{cycle}$$

5. Le synthétiseur de fonction de rang prouve que ρ_1 est bien fondé. De plus, il renvoie une relation de rang qui sur-approxime ρ_1 :

$$T_1 = x^1 < y^1 \wedge x^1 \geq x^0 + 1$$

6. À la fin de cette première itération, nous avons donc :

$$T = (x^1 < y^1 \wedge x^1 \geq x^0 + 1) \vee \texttt{false}$$

Deuxième itération :

$T = (x^1 < y^1 \wedge x^1 \geq x^0 + 1) \vee \texttt{false}$

1. Vérifier la validité de T à l'aide de l'analyse d'accessibilité binaire. L'analyse d'accessibilité binaire crée un nouveau programme 5.4 et y effectue l'analyse d'accessibilité.

Listing 5.4 Exemple d'application de TERMINATOR, deuxième itération $\widehat{P}$

```
1       int x := nondet(), y := nondet(), z := nondet()
2       if (y > 0)
3               do {
3.1                     if ('pc == 5)
3.2                             if (not ((x < y ∧ x ≥' x)
                                        ∨ false))
3.3                     if ('pc == 0)
3.4                             if (nondet())
3.5                                     'x := x
3.6                                     'y := y
3.7                                     'z := z
3.8                                     'pc := 5
4                       if (nondet())
5                               x := x + y
6                       else
7                               z := x − y
8               } while (x < y ∧ y < z)
```

2. Contre-exemple atteignant la déclaration `ERROR` :

$$1 \to 2 \to 3 \to 3.1 \to 3.3 \to 3.4 \to 3.5 \to 3.6 \to 3.7 \to 3.8$$

$$\to 4 \to 6 \to 7 \to 8$$

3. Ce contre-exemple peut être séparé en une *tige* et un *cycle* :

$$\begin{array}{rcl} \mathit{tige} & = & 1 \to 2 \to 3 \\ \mathit{cycle} & = & 4 \to 6 \to 7 \to 8 \to 3 \end{array}$$

   On obtient donc les relations suivantes :

$$\begin{array}{rcl} R_{tige}((x^0,y^0,z^0),(x^1,y^1,z^1)) & \triangleq & y^1 > 0 \wedge x^1 = x^0 \wedge z^1 = z^0 \\ R_{cycle}((x^0,y^0,z^0),(x^1,y^1,z^1)) & \triangleq & z^1 = x^0 - y^0 \wedge x^1 = x^0 \\ & & \wedge y^1 = y^0 \wedge x^1 < y^1 \\ & & \wedge y^1 < z^1 \end{array}$$

4. On appelle un synthétiseur de fonction de rang, avec le contre-exemple ρ_2

$$\rho_2(s_1,s_2) \triangleq \exists s_0 \in I.(s_0,s_1) \in R_{tige} \wedge (s_1,s_2) \in R_{cycle}$$

5. Le synthétiseur de fonction de rang prouve que ρ_2 est bien fondé. De plus, il renvoie une relation de rang qui sur-approxime ρ_2 :

$$T_2 = y^1 < z^1 \wedge z^1 \geq z^0 - 1$$

6. À la fin de cette deuxième itération, nous avons donc :

$$T = (y^1 < z^1 \wedge z^1 \geq z^0 - 1) \vee (x^1 < y^1 \wedge x^1 \geq x^0 + 1) \vee \texttt{false}$$

Troisième itération :

$T = (y^1 < z^1 \wedge z^1 \geq z^0 - 1) \vee (x^1 < y^1 \wedge x^1 \geq x^0 + 1) \vee \texttt{false}$

1. Vérifier la validité de T à l'aide de l'analyse d'accessibilité binaire. L'analyse d'accessibilité binaire crée un nouveau programme 5.5 et y effectue l'analyse d'accessibilité.

Listing 5.5 Exemple d'application de TERMINATOR, troisième itération $\widehat{P}$

```
1       int x := nondet(), y := nondet(), z := nondet()
2       if (y > 0) {
3               do {
3.1                     if ('pc == 5)
3.2                             if (not ((y < z ∧ z ≤' z)
3.3                                     ∨ (x < y ∧ x ≥' x)
3.4                                     ∨ false))
3.5                     if ('pc == 0)
3.6                             if (nondet())
3.7                                     'x := x
3.8                                     'y := y
3.9                                     'z := z
3.10                                    'pc := 5
4                       if (nondet())
5                               x := x + y
6                       else
7                               z := x − y
8               } while (x < y ∧ y < z)
```

2. Le point de programme contenant l'erreur n'est pas atteignable. L'argument de terminaison final est donc :

$$T = (y^1 < z^1 \wedge z^1 \geq z^0 - 1) \vee (x^1 < y^1 \wedge x^1 \geq x^0 + 1) \vee \texttt{false}$$

En conclusion, étant donné que le point de programme contenant l'erreur n'est pas atteignable, alors l'inclusion $R_I^+ \subseteq T$ est valide. Grâce au fait que cette inclusion soit prouvée valide, nous pouvons conclure que le programme se termine.

5.2 Analyse variante à partir d'analyse invariante

Cette section présente des analyses d'assertions variantes, pour prouver la terminaison d'un programme ainsi que des propriétés de vivacité. Ces analyses sont induites d'analyses d'invariances. En fonction de l'analyse d'invariance sous-jacente choisie, on obtiendra une analyse de variance différente. Ces analyses ont été développées par Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano et Peter O'Hearn en 2007 [3].

Tout d'abord, une mise en contexte général de cette méthode sera donnée à la section 5.2.1. Ensuite, dans le but de comprendre cette méthode par l'exemple, un exemple informel sera présenté à la section 5.2.2. Suite à cela, l'algorithme formel utilisé sera décrit à la section 5.2.3. Enfin, un exemple complet d'application de cet algorithme sera présenté à la section 5.2.4.

5.2.1 Mise en contexte

Ces analyses de variances donnent lieu à de nouveaux démonstrateurs de terminaison qui sont tout à fait compétitifs avec les démonstrateurs actuels.

La méthode d'analyse variante utilise un vocabulaire précis, commençons donc par quelques définitions :

Assertion invariante pour un point de programme l : déclaration qui reste toujours valide au point l durant l'exécution du programme.

Assertion variante pour un point de programme l : déclaration qui est valide d'un état au point l à un ancien état qui était aussi au point l.

Analyse invariante : prend un programme en entrée et retourne un ensemble d'invariants (assertion invariante disjointe) qui sont indexés par des points de programme. Chaque point de programme a un invariant. Ces invariants peuvent être directement utilisés pour prouver des propriétés de sécurité ou indirectement pour aider la construction de relation de transition abstraite.

Analyse variante : déduit des assertions variantes qui permettent de prouver la terminaison et les propriétés de vivacité.

Les assertions variantes déduites pendant l'analyse peuvent être utilisées pour établir des prédicats de terminaison locale.

— Si ces prédicats peuvent être établis pour chaque assertion variante d'un programme, alors il est prouvé que le programme se termine.
L'exactitude de cette étape se base sur l'article « Transition Invariant » de Podelski et Rybalchenko [16] où il montre un résultat sur les surapproximations disjointement bien fondées.

— Si le prédicat est établi pour seulement un sous-ensemble des assertions variantes, cela signifie qu'une propriété de vivacité différente est valide pour le programme.

5.2.2 Exemple informel

Cette sous-section décrit un exemple informel facile à comprendre dans le but de saisir l'essentiel de la méthode.

1. Considérons le programme 5.6.

Listing 5.6 Premier exemple pour illustrer l'analyse variante

```
1  x ∈ ℤ
2  while x > 0
3          x := x − 1
```

2. Transformons le programme considéré. Il devient le programme 5.7

Listing 5.7 Premier exemple transformé pour illustrer l'analyse variante

```
1  int x
2  int x0 := x
3  if x > 0
4          x := x − 1
5  else
6          stop
7  while x > 0
8          x := x − 1
```

Notons que le $x0$ à la deuxième ligne du programme sert uniquement à définir un invariant pour la suite de cet exemple.

3. Analysons l'invariant au point de programme du début de la ligne 7 :

$$(x \geq 0) \wedge (x0 \geq 1 + x)$$

Cet invariant est une sur-approximation de la fermeture transitive du programme, notée R^+.
On obtient ce résultat grâce à la transformation effectuée à l'étape 2. En effet, on est passé au moins une fois par le corps de la boucle à la ligne 4.

4. L'invariant trouvé au point de programme du début de la ligne 7 est une disjonction finie d'invariants bien fondés, réduite à une conjonction bien fondée.
On peut donc conclure que la boucle se termine.

5.2.3 Méthode d'analyse variante

L'algorithme 5.8 est l'analyse variante. Il faut instancier cet algorithme à un domaine particulier. Par exemple, le domaine abstrait Octagon [14] ou encore Polyhedra [11].
Pour instancier l'algorithme, il faut fournir l'implémentation des composants suivants :

`InvarianceAnalysis` : l'analyse invariance sous-jacente

`Step` : une seule étape de `InvarianceAnalysis`

`Seed` : une opération additionnelle sur les éléments du domaine abstrait

`WellFounded` : une opération additionnelle sur les éléments du domaine abstrait

Listing 5.8 Algorithme paramétrisé d'analyse variante

```
1  input
2          P : programme à analyser
3          L : ensemble de points de coupure du programme
4          I# : ensemble d'états initiaux
5  output
6          ensemble des prédicats de terminaison locale
7  begin
8          IAs := InvarianceAnalysis(P, I#)
9          foreach l ∈ L do
10                 LTPreds[l] := true
11                 O := Isolate(P, L, l)
12                 foreach q ∈ IAs tel que pc(q) = l do
13                         VAs:=InvarianceAnalysis(O, Step(O, {Seed(q)}))
14                         foreach r ∈ VAs do
15                                 if pc(r) = l ∧ ¬WellFounded(r)
16                                         LTPrefs[l] := false
17         return LTPrefs
18 end.
```

L'algorithme effectue les étapes suivantes :

1. Exécuter l'analyse invariante, générant un ensemble d'assertions invariantes, *IAs*.
2. Convertir chaque élément q (de *IAs*) en relations binaires, via l'opération `Seed`.
3. Exécuter une étape de l'analyse d'invariance dans le but de générer un point fixe sur les assertions variantes, *VAs*.
   - Cette étape permet de générer une approximation des comportements de la boucle.
   - L'étape faite avant la réexécution de l'analyse d'invariance permet de mettre à profit le résultat de l'article « Transition Invariant » [16] sur le bon fondement d'une relation à une sur-approximation

de sa fermeture transitive non réflexive. Sans `Step`, on obtiendrait la fermeture transitive réflexive.

4. Tester la validité de l'ensemble des prédicats de terminaison locale pour chaque élément de *VAs*, grâce à l'opération `WellFounded`.

Un prédicat de terminaison d'un point de programme l est valide si $LTPreds[l] = \texttt{true}$.

5.2.4 Exemple complet

Pour illustrer cette méthode, nous allons l'appliquer sur le programme 5.9.

Listing 5.9 Deuxième exemple pour illustrer l'analyse variante

```
1  while (nondet()) {
2          while (x > a ∧ y > b) {
3                  if (nondet()) {
4                          do {
5                                  x := x − 1
6                          } while (x > 10)
7                  } else {
8                          y := y − 1
9                  }
10         }
11 }
```

Quelques remarques :

— `nondet()` est un choix non déterministe

— L'analyse invariante sous-jacente est basée sur le domaine Octagon. Ce domaine permet d'exprimer des conjonctions d'inégalités de la forme $\pm x + \pm y \leq c$.
x et y étant des variables et c une constante.

— Une assertion est valide à un point de programme l si et seulement si elle est toujours valide au début de la ligne l, avant d'exécuter l'instruction de cette ligne.

Déroulement de l'algorithme :

1. Préliminaires :
   (a) Tout d'abord, nous devons choisir des points de coupure pour les boucles du programme. Un point de coupure d'une boucle est le numéro de la ligne d'une instruction que le programme n'effectuera pas dans le cas où le test de la boucle n'est pas valide. L'ensemble L contient les points de coupures choisis :

$$L = \{2, 3, 5\}$$

2 : point de coupure pour la boucle des lignes 1 à 11

3 : point de coupure pour la boucle des lignes 2 à 10

5 : point de coupure pour la boucle des lignes 4 à 6

(b) Définitions des prédicats de terminaison locale pour chaque point de coupure :

La ligne 2 est visitée infiniment souvent si l'exécution du programme quitte la boucle des lignes 1 à 11 infiniment souvent.

La ligne 3 est visitée infiniment souvent si l'exécution du programme quitte la boucle des lignes 2 à 10 infiniment souvent.

La ligne 5 est visitée infiniment souvent si l'exécution du programme quitte la boucle des lignes 4 à 6 infiniment souvent.

$\rightarrow$ du fait que la boucle externe (ligne 1 à 8) ne se termine pas, l'analyse de variance ne va pas pouvoir prouver le prédicat de la ligne 2. Cependant, le fait d'utiliser un programme à boucles imbriquées permet de montrer la modularité offerte par les prédicats de terminaison locale. On pourra tout de même montrer la validité des prédicats des boucles internes.

2. Tout d'abord, l'analyse invariante est exécutée (ligne 8 de l'algorithme 5.8)
   On part de l'état initial :

$$I^{\#} = (pc = 1 \wedge x \geq a + 1 \wedge y \geq b + 1)$$

   De $I^{\#}$, l'analyse invariante peut générer un invariant $IA_3 \in IAs$:

$$IA_3 \triangleq pc = 3 \wedge x \geq a + 1 \wedge y \geq b + 1$$

3. Ensuite, l'isolation est réalisée (ligne 11 de l'algorithme 5.8)
   On va isoler le plus petit sous-graphe fortement connecté de P contenant le point de programme 3 :

   (a) À partir du programme P, on construit un nouveau programme O.
   O est une copie de P où on rajoute une ligne 9 contenant :

$$\}\texttt{assume}(\texttt{false})$$

   Grâce à `assume`, les exécutions qui permettent de quitter la boucle ne sont pas considérées.

   (b) L'état initial de O est 3.

   Cette étape d'isolation permet d'analyser des boucles internes. Il permet d'établir des prédicats de terminaison locale imbriqués dans une boucle qui diverge.

4. Maintenant, les assertions variantes peuvent être déduites (lignes 12 et 13 de l'algorithme 5.8)
   L'analyse invariante va être utilisée pour raisonner sur le programme isolé O.
   (a) On prend toutes les disjonctions de l'assertion invariante du point de programme 3 et on les convertit en relations binaires d'états à états :

$$\begin{array}{rcl} \texttt{Seed}(IA_3) & = & (pc = 3 \wedge pc_s = 3 \\ & & \wedge x \geq a + 1 \wedge y \geq b + 1 \\ & & \wedge x_s = x \wedge y_s = y \\ & & \wedge a_s = a \wedge b_s = b) \end{array}$$

   L'état s donne des valeurs aux variables $\{pc_s, x_s, y_s, a_s, b_s\}$ et l'état t donne des valeurs aux variables $\{pc, x, y, a, b\}$

   (b) On fait une étape à partir de $\texttt{Seed}(IA_3)$ avec `Step`, approximativement une étape de la sémantique du programme, nous donne :

$$pc_s = 3 \wedge pc = 4 \wedge x \geq a+1 \wedge y \geq b+1 \wedge x_s = x \wedge y_s = y \wedge a_s = a \wedge b_s = b$$

   (c) On exécute `InvarianceAnalysis` sur le programme O, avec pc_s comme état initial. Cela donne un ensemble d'invariants aux points de programmes 2, 3, etc. qui correspondent aux VAs dans l'algorithme `VarianceAnalysis`.

$$\begin{array}{rcl} VA_3^A & \triangleq & (pc_s = 3 \wedge pc = 3 \\ & & \wedge x \geq a + 1 \wedge y \geq b + 1 \\ & & \wedge x_s = x + 1 \wedge y_s = y \\ & & \wedge a_s = a \wedge b_s = b) \\ VA_3^B & \triangleq & (pc_s = 3 \wedge pc = 3 \\ & & \wedge x \geq a + 1 \wedge y \geq b + 1 \\ & & \wedge x_s = x \wedge y_s = y + 1 \\ & & \wedge a_s = a \wedge b_s = b) \\ VA_3^C & \triangleq & (pc_s = 3 \wedge pc = 3 \\ & & \wedge x \geq a - 1 \wedge y \geq b + 1 \\ & & \wedge x_s = x + 1 \wedge y_s = y + 1 \\ & & \wedge a_s = a \wedge b_s = b) \end{array}$$

$$\{ VA_3^A, VA_3^B, VA_3^C \} \subseteq VAs$$

   L'union de ces trois relations $VA_3^A \vee VA_3^B \vee VA_3^C$ forme une assertion variante pour la ligne 3 du programme P, qui reste un sur-ensemble des transitions possibles d'un état au point de programme 3 à un autre état aussi au point de programme 3.
   La disjonction $VA_3^A \vee VA_3^B \vee VA_3^C$ est un sur-ensemble de la fermeture transitive de la relation de transition du programme restreint aux paires d'états atteignables aux points de programme 3.

5. Finalement, la preuve des prédicats de terminaison locale peut être établie (lignes 14 à 17 de l'algorithme 5.8)
   - On utilise l'assertion variante de la ligne 3 dans O pour établir le prédicat de terminaison locale de la ligne 3 dans P.
   - Considérons la relation suivante :

     $$Tr_3 = \{(s,t) \mid s \models IA_3 \wedge s \longrightarrow_O^+ t \wedge t(pc) = 3\}$$

     $\longrightarrow_O$: représente la relation de transition de O.
   - En montrant que Tr_3 est bien fondée, on peut déduire le prédicat de terminaison locale suivant : « Le point de programme 3 n'est pas visité infiniment souvent en exécutant le sous-programme isolé O. »
   - Pour rappel : une relation R est bien fondée si et seulement si sa fermeture transitive R^+ est un sous-ensemble de l'union finie $T_1 \cup \cdots \cup T_n$ et chaque relation T_i est bien fondée.
   - On sait que $VA_3^A \cup VA_3^B \cup VA_3^C$ sur-approxime la fermeture transitive du programme P limité aux états du point de programme 3. De plus, étant donné que chaque relation est bien fondée, on peut garantir que le programme ne va pas visiter le point de programme 3 infiniment souvent, sauf s'il quitte le sous-programme infiniment souvent.
   - Enfin, on peut prouver que chacune des relations VA_3^A, VA_3^B et VA_3^C sont bien fondées. Du fait que ces relations sont représentées comme une conjonction d'inégalités linéaires, elles peuvent être automatiquement prouvées par un synthétiseur de fonction de rang.

En conclusion, bien qu'il est impossible de prouver la terminaison de la boucle, la méthode a permis de prouver la terminaison d'une boucle imbriquée.

5.3 Algorithme d'abstraction abstraite

Cette section présente un algorithme d'abstraction abstraite pour démontrer qu'un programme finit pour toutes ses entrées possibles. Cet algorithme utilise un nouveau domaine abstrait qui utilise des relations de rang pour conserver les relations entre les états du programme. Il a été développé par Aziam Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, et Hongseok Yang en 2008 [5].

Tout d'abord, une mise en contexte général de cet algorithme sera donnée à la section 5.3.1. Ensuite, un exemple informel sera présenté à la section 5.3.2. Enfin, le domaine abstrait utilisé sera présenté à la section 5.3.3.

5.3.1 Mise en contexte

L'algorithme d'abstraction abstraite suit le travail de l'article « Variance analyses from invariance analyses » [3] et celui de l'article « Termination proofs for systems code » [9]. Il emploie le principe de disjonction bien fondée, de l'article « Transition invariants » [16], dans le but de séparer l'argument de terminaison en de multiples relations de rang correspondant à différentes situations du programme.

L'article « Transition Invariant » [16] établit la relation $\tau^+ \subseteq \bigcup_{i=1}^{n} r_i$, mais ne répond pas aux deux problèmes suivants :

1. Quel est le meilleur moyen pour trouver les fonctions de rang $r_1, r_2, \ldots, r_n$?
2. Comment vérifier la condition suivante ?

$$\tau^+ \subseteq \bigcup_{i=1}^{n} r_i$$

On peut résoudre cette question par n'importe quel interpréteur abstrait, mais cela est coûteux. De plus son imprécision peut le mener à échouer.

Cet algorithme résoud deux problèmes d'une nouvelle manière :

— On trouve les fonctions de rang en développant un domaine abstrait qui est paramétrisé par des ensembles de fonctions de rang. La signification de chacune des fonctions de rang surestime la relation entre les états intermédiaires du programme.

— On emploie des générations itératives de point fixe pour générer un ensemble de fonctions de rang ou pour déterminer que le programme peut diverger.

— Le synthétiseur de rang est invoqué avec des relations de plus en plus larges, obtenues en composant l'approximation courante avec chaque contrainte possible.
Appeler le synthétiseur de rang permet d'abstraire des informations non nécessaires à la terminaison, mais qui maintiennent une distinction entre les différentes fonctions de rang.

— Lorsqu'un point fixe est atteint, l'équation $\tau^+ \subseteq \bigcup_{i=1}^{n} r_i$ est garantie et donc il n'est pas nécessaire d'effectuer le test d'inclusion.

En d'autres mots :

1. L'argument de terminaison est construit en suivant la structure de la relation de transition $R = R_1 \cup \cdots \cup R_m$, en utilisant une procédure de synthèse de fonction de rang, qui est utilisée pour générer une approximation bien fondée $WF(X)$, d'une relation binaire X.
2. Le candidat initial $T = WF(R_1) \cup \cdots \cup WF(R_m)$ est étendu avec $WF(WF(R_i) \circ R_j)$. Cela continue jusqu'à ce que le point fixe soit atteint.

5.3.2 Exemple informel

Pour illustrer la méthode, nous allons l'appliquer sur le programme 5.10.

Listing 5.10 Programme d'exemple d'algorithme d'abstraction abstraite

```
1  while (x > 0 ∧ y > 0)
2          if (nondet())
3                  x := x − 1
4                  y := nondet()
5          else
6                  y := y − 1
```

R représente la relation de transition du corps de la boucle :

$$\begin{array}{rcl} R & \triangleq & C_1 \vee C_2 \\ C_1 & \triangleq & x^{-1} > 0 \wedge y^{-1} > 0 \wedge x^0 = x^{-1} - 1 \\ C_2 & \triangleq & x^{-1} > 0 \wedge y^{-1} > 0 \wedge x^0 = x^{-1} \wedge y^0 = y^{-1} - 1 \end{array}$$

Application de la méthode :

1. On prend chaque disjonction de R et on y effectue une synthèse de fonction de rang.
   On obtient donc :
   $$\begin{array}{rcl} \mathrm{RFS}(C_1) & = & x \\ \mathrm{RFS}(C_2) & = & y \end{array}$$
   L'abréviation RFS provient des termes anglais « Rank Function Synthesis » qui signifie « synthèse de fonction de rang ».

2. Pour chaque disjonction, on calcule un ensemble de variables dont les valeurs ne changent pas. Dans notre cas, C_1 peut modifier x et y et C_2 seulement y.
   Pour une facilité de lecture, nous introduisons les variables suivantes :

$$\begin{array}{lcl} T_x & = & x^{-1} \geq 0 \wedge x^{-1} - 1 \geq x^0 \\ T_y & = & y^{-1} \geq 0 \wedge y^{-1} - 1 \geq y^0 \\ V_x & = & x^{-1} = x^0 \end{array}$$

   De ce fait, l'état initial abstrait est :

$$A_0 = T_x \vee [T_y \wedge V_x]$$

   A_0 sur-approxime le corps de la boucle de R :

$$R \subseteq A_0$$

3. On génère l'état abstrait suivant, A_1.
   A_1 sur-approxime la composition relationnelle de A_0 et R.
   Pour cela, il faut :
   (a) prendre chaque disjonction de A_0, et chaque disjonction de R ;
   (b) les composer ;
   (c) y effectuer RFS, la synthèse de fonction de rang ;
   (d) déduire les variables qui ne se modifient pas ;
   (e) construire l'union de la nouvelle fonction de rang avec A_0

   Nous obtenons :

$$\begin{array}{lcllll} A_1 & = & A_0 \circ R & & & \\ & \overset{(a)}{=} & (T_x \vee [T_y \wedge V_x]) \circ (C_1 \vee C_2) & & & \\ & \overset{(b)}{=} & (T_x \circ C_1) & \vee([T_y \wedge V_x] \circ C_1) & \vee(T_x \circ C_2) & \vee([T_y \wedge V_x] \circ C_2) \\ & \overset{(c)}{=} & T_x & \vee T_x & \vee T_x & \vee T_y \\ & \overset{(d)}{=} & \emptyset & \vee \emptyset & \vee \emptyset & \vee V_x \\ & \overset{(e)}{=} & T_x \vee [T_y \wedge V_x] & & & \end{array}$$

   On en déduit que :

$$T_x \vee [T_y \wedge V_x] \subseteq A_0$$

   De ce fait :

$$A \circ R \subseteq A_0$$

4. A_0 est un point fixe et sur-approxime R. On peut conclure la terminaison de deux manières différentes :

(a) A_0 est une disjonction de deux relations bien fondées

(b) A_0 est un invariant de transition inductif car

$$\begin{cases} R & \subseteq & A_0 \\ A_0 \circ R & \subseteq & A_0 \end{cases}$$

Nous pouvons conclure que R est bien fondée et de ce fait nous savons que le programme se terminera.

5.3.3 Domaine abstrait

L'analyse est paramétrisée par un domaine pour représenter des relations sur des états.
St représente la mémoire, c'est-à-dire l'association de chaque nom de variable à sa valeur :

$$\mathsf{St} = \text{Vars} \to \text{Real}$$

Le domaine est spécifié par les données suivantes :

1. Un ensemble D et une fonction monotone $\gamma_r : D :\to \mathcal{P}(\mathsf{St} \times \mathsf{St})$
2. Un élément d'identité abstrait d_{id} dans D, qui satisfait

   $$\Delta_{\mathsf{St}} \subseteq \gamma_r(d_{\mathsf{id}})$$

   où Δ_{St} est la relation d'identité sur St.
3. Un opérateur RFS : $D \to \mathcal{P}_{\mathsf{fin}}(D) \uplus \{\top\}$. Cet opérateur synthétise des fonctions de rang.
   (a) RFS génère une sur-approximation :

   $$\text{RFS}(d) \neq \top \Rightarrow \gamma_r(d) \subseteq \bigcup \{\gamma_r(d') \mid d' \in \text{RFS}(d)\}.$$

   (b) RFS(d) décrit une relation bien fondée :

   $$\text{RFS}(d) \neq \top \Rightarrow \bigcup \{\gamma_r(d') \mid d' \in \text{RFS}(d)\} \text{ est bien fondée.}$$

4. Une fonction de transfert abstraite $\texttt{trans}(a)$ pour chaque déclaration a.
   La fonction $\texttt{trans}(a)$ est de type : $D \to \mathcal{P}_{\mathsf{fin}}(D)$, et satisfait

   $$\forall d \in D(\gamma_r(d); [\![a]\!]) \subseteq \bigcup \{\gamma_r(d') \mid d' \in \texttt{trans}(a)(d)\}$$

5. Un opérateur de composition abstrait $\texttt{comp} : D \times D \to D$ tel que

   $$\gamma_r(d); \gamma_r(d') \subseteq \gamma_r(\texttt{comp}(d, d')).$$

Le domaine abstrait $\mathcal{A}$ de l'analyseur est

$$\mathcal{A} \triangleq (\mathcal{P}_{\mathsf{fin}}(D))^{\top}$$

Il est ordonné par $\sqsubseteq$, étendu avec $\top$. Nous avons donc $A \sqsubseteq A'$ si et seulement si

$$(A' = \top) \vee (A, A' \in \mathcal{P}_{\mathsf{fin}}(D) \wedge A \subseteq A')$$

— Chaque élément A dans $\mathcal{A}$ représente un ensemble de séquences d'états finies ou infinies. On les appelle des traces.

— L'élément $\top$ représente l'ensemble de toutes les traces, incluant celles qui ne se finissent pas.

— Les éléments non$-\top$ représentent un ensemble de traces non vides qui se finissent. Les états initiaux et finaux sont représentés par d.

— $\gamma_r(A) = \bigcup\{\gamma_r(d) \mid d \in A\}$, la disjonction de d dans A, et définit $\mathcal{T}$ un ensemble de traces non vides :

$$\mathcal{T} \triangleq \mathsf{St}^{+} \cup \mathsf{St}^{\infty}$$

La définition formelle de A est donnée par la fonction de concrétisation γ :

$$\begin{array}{rcl} \gamma & : & \mathcal{A} \to \mathcal{P}(\mathcal{T}) \\ \gamma(A) & \triangleq & \texttt{if}(A = \top) \\ & & \texttt{then}\ \mathcal{T} \\ & & \texttt{else}\ \{\tau \mid \tau \text{ n'est pas vide, et } \tau_0[\gamma_r(A)]\tau_{|\tau|-1}\} \end{array}$$

où

— $|\tau|$ est la longueur de la trace τ

— τ_n est le $n^{\text{ième}}$ état de la trace τ

— $s[r]s'$ est la relation r entre s et s'

En pratique, la méthode utilise un domaine disjoint de relation de rang lié à des informations sur les variables inchangées. On obtient donc des relations disjointes de la forme $T_e \wedge V_X$, où :

$$\begin{array}{rcl} V_X & \triangleq & \bigwedge_{x \in X} x^{-i} = x^0 \\ T_e & \triangleq & e^{-i} \geq 0 \wedge e^{-i} - 1 \geq e^0 \end{array}$$

Remarques :

— e est une expression contenant des variables x ;

— x^{-i} représente les valeurs des variables x de i états précédents x^0

En conclusion de cette section, voici quelques remarques :

— Le démonstrateur de terminaison développé dans cette section est très efficace pour prouver des programmes impératifs. En particulier, il est plus rapide que TERMINATOR [9] et que les analyses variantes basées sur Octagon ou Polyhedra [3].

— L'analyse variante utilisée dans cette section utilise directement des fonctions de rang. Tandis que l'analyse variante de [3] utilise des fonctions de rang qui sont construites par des domaines existants pour l'invariance.

— Les relations de rang sont directement abstraites. Cela permet d'être plus précis.

— Les fonctions de rang sont générées itérativement, contrairement à Terminator [9] qui utilise des contre-exemples pour les rendre plus précises.

Conclusion

Nous avons commencé par établir une catégorisation des méthodes permettant de prouver la terminaison de boucles. Nous nous sommes familiarisés avec ce domaine en appliquant une méthode intuitive sur des exemples. Les deux autres catégories étudiées sont les méthodes traditionnelles et les méthodes modernes.

Nous avons étudiés deux des méthodes actuelles qui se basent sur les méthodes traditionnelles et deux autres utilisant des principes non repris dans cette catégorie. Cela nous a permis de nous rendre compte des différentes techniques possibles pour atteindre un même et unique objectif : prouver la terminaison d'une boucle.

Ensuite, nous avons détaillé la méthode de Chen ainsi que son algorithme général. Nous avons implémenté cette méthode et elle a été appliquée sur différents exemples.

Enfin, nous avons étudié trois récents démonstrateurs. Tous se basent sur le même principe utilisé dans la catégorie des méthodes modernes. Ce sont des démonstrateurs qui se basent sur des arguments de terminaison disjonctifs.

Nous constatons que chaque méthode a ses avantages, son domaine d'application, mais aussi ses limites. Au fil du temps, les nouvelles méthodes se basent sur les anciennes et essayent de les améliorer ou de les étendre.

Une partie non négligeable de ce mémoire a été accordée à l'étude de la méthode de Chen décrite dans son article [7]. L'étude de cette méthode a été approfondie, car c'est une des plus performante actuellement.

Rappelons deux propriétés importantes de cette méthode :

— La méthode détermine seulement les boucles terminantes.
Ceci s'explique par la définition du terme « méthode ». En effet, une méthode n'est pas un algorithme. Un algorithme doit toujours se terminer, alors qu'une méthode tente de s'approcher d'un résultat. Cette méthode est implémentée par un algorithme borné. Après un certain nombre défini d'itérations, l'algorithme s'arrête si une disjonction de fonction de rang valide n'a toujours pas été trouvée.

— La méthode est correcte, mais on ne sait pas si elle est complète.
Pour rappel, la méthode de Podeslski & Rybalchenko sait prouver toutes les boucles ayant une fonction de rang linéaire. Elle est donc complète. Étant donné que la méthode de Chen utilise l'algorithme de Podelski & Rybalchenko, elle est de ce fait également complète pour la classe de boucles ayant une fonction de rang linéaire. Cependant, la méthode de Podelski & Rybalchenko n'est pas capable de prouver une boucle ayant une fonction de rang non linéaire. Grâce à la disjonction des fonctions de rang, la méthode de Chen en est capable, mais sa complétude n'a pas été prouvée pour cette classe de boucles.

Comment pourrions-nous améliorer la méthode de Chen ?

La méthode de Chen se base sur l'algorithme de Podelski & Rybalchenko. Cet algorithme prouve les programmes à fonction de rang linéaire. Les fonctions de rang linéaires sont une sous-classe des fonctions de rang linéaires à terme.

Une amélioration possible de la méthode de Chen serait de remplacer l'utilisation de l'algorithme de Podelski & Rybalchenko par l'algorithme des fonctions de rang linéaires à terme.

La classe de boucles couverte par la méthode de Chen n'étant pas définie, il est difficile de prévoir le résultat d'un tel changement. Cependant, par intuition, cela donnera naissance à une méthode capable de prouver une plus large classe de boucles.

Cette matière reste à ce jour en pleine évolution à la recherche d'une méthode plus générale et applicable dans la majorité des cas. Rappelons que le problème de terminaison étant un problème indécidable, il ne sera jamais possible de trouver une méthode permettant de prouver tous les programmes possibles.

Nous espérons toutefois que ce mémoire par la description des différentes méthodes et les exemples d'applications contribuera à une vision de l'état de l'art actuel et permettra de mieux saisir les opportunités qu'offrent ces méthodes ainsi que leur mise en pratique.

Bibliographie

[1] Z3. http://z3.codeplex.com.

[2] Roberto Bagnara and Fred Mesnard. Eventual Linear Ranking Functions. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 229–238, New York, NY, USA, 2013. ACM.

[3] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance Analyses from Invariance Analyses. *SIGPLAN Not.*, 42(1) :211–224, January 2007.

[4] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The Polyranking Principle. In *Proceedings of the 32Nd International Conference on Automata, Languages and Programming*, ICALP'05, pages 1349–1361, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking Abstractions. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 148–162. Springer Berlin Heidelberg, 2008.

[6] Hong Yi Chen, Shaked Flur, and Supratik Mukhopadhyay. LSL Test Suite.

[7] Hong Yi Chen, Shaked Flur, and Supratik Mukhopadhyay. Termination Proofs for Linear Simple Loops. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 422–438, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Lothar Collatz. The Collatz Conjecture. 1937.

[9] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination Proofs for Systems Code. *SIGPLAN Not.*, 41(6) :415–426, june 2006.

[10] Patrick Cousot and Radhia Cousot. An Abstract Interpretation Framework for Termination. *SIGPLAN Not.*, 47(1) :245–258, January 2012.

[11] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[12] Robert W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19 :19–32, 1967.

[13] R.K. Guy. In *Unsolved Problems in Number Theory*, pages 215–218, Problem E16. Springer, Second edition, 1994.

[14] Antoine Miné. The Octagon Abstract Domain. *Higher Order Symbol. Comput.*, 19(1) :31–100, March 2006.

[15] Andreas Podelski and Andrey Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. 2937 :239–251, 2004.

[16] Andreas Podelski and Andrey Rybalchenko. Transition Invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, LICS '04, pages 32–41, Washington, DC, USA, 2004. IEEE Computer Society.

[17] F. P. Ramsey. On a Problem in Formal Logic. *Proc. London Math. Soc. (3)*, 30 :264–286, 1930.

[18] Alan Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. London Mathematical Society, 1936.

[19] Alan Turing. Checking a Large Routine. In *report of a Conference on High Speed automatic Calculating Machines*, 1949.

[20] Caterina Urban. Piecewise-Defined Ranking Functions. In Johannes Waldmann, editor, *13th International Workshop on Termination*, pages 69–73, Bertinoro, Italy, Aug 2013.

[21] Caterina Urban. The Abstract Domain of Segmented Ranking Functions. In *SAS*, pages 43–62, 2013.

Annexes

A Code source et résultats

Ces différentes annexes ont été utilisées dans le chapitre 4. Ces codes sources ont permis d'éviter de longs calculs manuscrits.

- L'annexe A.1 calcule un système d'inéquations exprimé sous forme matricielle pour les méthodes de Podelski & Rybalchenko normale et étendue dans le langage de programmation *Scheme*.
- L'annexe A.2 reprend les résultats générés en interprétant le code source de l'annexe A.1.
- L'annexe A.3 résoud les systèmes d'inéquations de l'annexe A.2 dans le langage de programmation *SWI-Prolog*.
- L'annexe A.4 teste la satisfiabilité de formule pour démontrer $R \subseteq T \wedge (T \circ R \subseteq T \vee R \circ T \subseteq T)$ par le solveur *Z3*.
- L'annexe A.5 reprend les résultats obtenus en exécutant le code source de l'annexe A.4.

A.1 Générer le système d'inéquations de la méthode de Podelski & Rybalchenko

```
#lang racket

(define (+list-string a . b)
  (add-+ (cons a b)))

(define (*string a b)
  (string-append (number->string a) "*" b))

(define (-string a b)
  (string-append "(" a "-" b ")"))

(define (matrix-multiply matrix1 matrix2)
  (map
```

```
   (lambda (row)
     (apply map
            (lambda column
              (apply + (map * row column)))
            matrix2))
   matrix1))

(define (matrix-add matrix1 matrix2)
  (map
   (lambda (x y)
     (map + x y)) matrix1 matrix2))

(define (matrix-string-sub matrix1 matrix2)
  (map
   (lambda (x y)
     (-string x y)) matrix1 matrix2))

(define (matrix-multiply-string matrix1 matrix2)
  (map
   (lambda (row)
     (apply map
            (lambda column
              (apply +list-string (map *string column
                 row)))
            matrix2))
   matrix1))

(define (make-list l n)
  (define (iter n)
    (if (= n 1)
        (list (string-append l "1"))
        (cons (string-append l (number->string n)) (
           iter (- n 1)))))
  (reverse (iter n)))

(define (add-+ list)
  (if (null? list)
      ""
      (string-append "+" (car list) (add-+ (cdr list))
         )))

(define (podelski A0 A1 b)
  (let ((eq1 (car (matrix-multiply-string (list (make-
     list "L1" (length A0))) A1)))
```

```
        (eq2 (car (matrix-multiply-string (list (
           matrix-string-sub (make-list "L1" (length
           A0)) (make-list "L2" (length A0)))) A0)))
        (eq3 (car (matrix-multiply-string (list (make-
           list "L2" (length A0))) (matrix-add A0 A1))
           ))
        (eq4 (car (matrix-multiply-string (list (make-
           list "L2" (length b))) b))))
    (define (display-eq eq)
      (unless (null? eq)
        (begin
          (display (car eq))
          (display "=0, ")
          (newline)
          (display-eq (cdr eq)))))
    (display-eq eq1)
    (display-eq eq2)
    (display-eq eq3)
    (display (car eq4))
    (display "<0")))

; i = 0, j = 1
(define (podelski-etendu A0 A1 A2 A12 b)
  (let ((eq1 (car (matrix-multiply-string (list (make-
     list "L1" (length A0))) A12)))
        (eq2 (car (matrix-multiply-string (list (make-
           list "L2" (length A0))) A2)))
        (eq3 (car (matrix-multiply-string (list (
           matrix-string-sub (make-list "L1" (length
           A0)) (make-list "L2" (length A0)))) A0)))
        (eq4 (car (matrix-multiply-string (list (make-
           list "L2" (length A0))) (matrix-add A0 A1))
           ))
        (eq5 (car (matrix-multiply-string (list (make-
           list "L2" (length b))) b))))
    (define (display-eq eq)
      (unless (null? eq)
        (begin
          (display (car eq))
          (display "=0, ")
          (newline)
          (display-eq (cdr eq)))))
    (display-eq eq1)
    (display-eq eq2)
```

```
    (display-eq eq3)
    (display-eq eq4)
    (display (car eq5))
    (display "<0")))

(define (print-matrix matrix)
  (unless (null? matrix)
    (display (car matrix))
    (newline)
    (print-matrix (cdr matrix))))

; PREMIER EXEMPLE (numero 1 dans l'article de Chen)
; L1.2 sous forme matricielle :
(define 1_A0 '((-1) (2) (-2) (1)))
(define 1_A1 '((0) (1) (-1) (-1)))
(define 1_b '((0) (10) (-10) (0)))

; L2.2 sous forme matricielle :
(define 1_A0_2 '((-1) (2) (-2) (0) (0) (0) (1) (0)))
(define 1_A1_2 '((0) (1) (-1) (-1) (2) (-2) (-1) (2)))
(define 1_A2_2 '((0) (0) (0) (0) (1) (-1) (0) (-1)))
(define 1_A12_2 '((0 0) (1 0) (-1 0) (-1 0) (2 1) (-2
   -1) (-1 0) (2 -1)))
(define 1_b_2 '((0) (10) (-10) (0) (10) (-10) (0) (0))
   )

; DEUXIEME EXEMPLE (numero 3 dans l'article de Chen)
; L sous forme matricielle :
(define 2_A0 '((-1) (-1) (1)))
(define 2_A1 '((0) (-2) (2)))
(define 2_b '((-1) (0) (0)))
(define 2_L1 '((0.6 0 0)))
(define 2_L2 '((1 0 0.3)))

; TROISIEME EXEMPLE (numero 21 dans l'article de Chen)
; L1.2 sous forme matricielle :
(define 3_A0 '((-1 0) (0 -1) (0 1) (0 -1) (0 1) (1 0))
   )
(define 3_A1 '((0 0) (1 0) (-1 0) (0 1) (0 -1) (-1 0))
   )
(define 3_b '((0) (0) (0) (-1) (1) (0)))
(define 3_L1 '((1 1 0 0 0 1)))
(define 3_L2 '((0 0 0 1 0 0)))
```

```
; QUATRIEME EXEMPLE (exemple de l'article Eventual
   Linear Ranking Functions)
; L sous forme matricielle :
(define 4_A0 '((-1 0) (0 -1) (-1 -1)))
(define 4_A1 '((0 0) (0 1) (1 0)))
(define 4_b '((0) (-1) (0)))

;L_synth sous forme matricielle :
(define 4_A0_2 '((-1 0) (0 -1) (-1 -1) (1 0)))
(define 4_A1_2 '((0 -1) (0 1) (1 0) (-1 0)))
(define 4_b_2 '((0) (-1) (0) (0)))
(define 4_L1_2 '((0 0 1 1)))
(define 4_L2_2 '((0 1 0 0)))


(define (generer-equation)
  (display "% Générer les équations")
  (newline)
  (display "% Premier exemple : \n")
  (display "% L1.2 : \n")
  (podelski 1_A0 1_A1 1_b)
  (newline)
  (display "% L2.2 : \n")
  (podelski-etendu 1_A0_2 1_A1_2 1_A2_2 1_A12_2 1_b_2)
  (newline)
  (newline)
  (display "% Deuxième exemple : \n")
  (display "% L :\n")
  (podelski 2_A0 2_A1 2_b)
  (newline)
  (newline)
  (display "% Troisieme exemple : \n")
  (display "% L :\n")
  (podelski 3_A0 3_A1 3_b)
  (newline)
  (newline)
  (display "% Quatrieme exemple : \n")
  (display "% L :\n")
  (podelski 4_A0 4_A1 4_b)
  (newline)
  (display "% Quatrieme exemple : \n")
  (display "% L_synth :\n")
  (podelski 4_A0_2 4_A1_2 4_b_2)
  (newline)
```

```
  (newline))

(define (calcul-lambda)
  (display "Calculer les lambdas")
  (newline)
  (display "Deuxième exemple")
  (newline)
  (display "L2 * A1 = ")
  (display (matrix-multiply 2_L2 2_A1))
  (newline)
  (display "L1 * b = ")
  (display (matrix-multiply 2_L1 2_b))
  (newline)
  (display "L2 * b = ")
  (display (matrix-multiply 2_L2 2_b))
  (newline)
  (newline)
  (display "Troisième exemple")
  (newline)
  (display "L2 * A1 = ")
  (display (matrix-multiply 3_L2 3_A1))
  (newline)
  (display "L1 * b = ")
  (display (matrix-multiply 3_L1 3_b))
  (newline)
  (display "L2 * b = ")
  (display (matrix-multiply 3_L2 3_b))
  (newline)
  (newline)
  (display "Quatrieme exemple")
  (newline)
  (display "L2 * A1 = ")
  (display (matrix-multiply 4_L2_2 4_A1_2))
  (newline)
  (display "L1 * b = ")
  (display (matrix-multiply 4_L1_2 4_b_2))
  (newline)
  (display "L2 * b = ")
  (display (matrix-multiply 4_L2_2 4_b_2))
  )

(generer-equation)

(calcul-lambda)
```

A.2 Systèmes d'inéquations générés par le code source de l'annexe A.1

```
% Générer les équations
% Premier exemple :
% L1.2 :
+0*L11+1*L12+-1*L13+-1*L14=0,
+-1*(L11-L21)+2*(L12-L22)+-2*(L13-L23)+1*(L14-L24)=0,
+-1*L21+3*L22+-3*L23+0*L24=0,
+0*L21+10*L22+-10*L23+0*L24<0
% L2.2 :
+0*L11+1*L12+-1*L13+-1*L14+2*L15+-2*L16+-1*L17+2*L18
    =0,
+0*L11+0*L12+0*L13+0*L14+1*L15+-1*L16+0*L17+-1*L18=0,
+0*L21+0*L22+0*L23+0*L24+1*L25+-1*L26+0*L27+-1*L28=0,
+-1*(L11-L21)+2*(L12-L22)+-2*(L13-L23)+0*(L14-L24)+0*(
    L15-L25)+0*(L16-L26)+1*(L17-L27)+0*(L18-L28)=0,
+-1*L21+3*L22+-3*L23+-1*L24+2*L25+-2*L26+0*L27+2*L28
    =0,
+0*L21+10*L22+-10*L23+0*L24+10*L25+-10*L26+0*L27+0*L28
    <0

% Deuxième exemple :
% L :
+0*L11+-2*L12+2*L13=0,
+-1*(L11-L21)+-1*(L12-L22)+1*(L13-L23)=0,
+-1*L21+-3*L22+3*L23=0,
+-1*L21+0*L22+0*L23<0

% Troisieme exemple :
% L :
+0*L11+1*L12+-1*L13+0*L14+0*L15+-1*L16=0,
+0*L11+0*L12+0*L13+1*L14+-1*L15+0*L16=0,
+-1*(L11-L21)+0*(L12-L22)+0*(L13-L23)+0*(L14-L24)+0*(
    L15-L25)+1*(L16-L26)=0,
+0*(L11-L21)+-1*(L12-L22)+1*(L13-L23)+-1*(L14-L24)+1*(
    L15-L25)+0*(L16-L26)=0,
+-1*L21+1*L22+-1*L23+0*L24+0*L25+0*L26=0,
+0*L21+-1*L22+1*L23+0*L24+0*L25+0*L26=0,
+0*L21+0*L22+0*L23+-1*L24+1*L25+0*L26<0

% Quatrieme exemple :
% L :
+0*L11+0*L12+1*L13=0,
```

```
+0*L11+1*L12+0*L13=0,
+-1*(L11-L21)+0*(L12-L22)+-1*(L13-L23)=0,
+0*(L11-L21)+-1*(L12-L22)+-1*(L13-L23)=0,
+-1*L21+0*L22+0*L23=0,
+0*L21+0*L22+-1*L23=0,
+0*L21+-1*L22+0*L23<0
% Quatrieme exemple :
% L_synth :
+0*L11+0*L12+1*L13+-1*L14=0,
+-1*L11+1*L12+0*L13+0*L14=0,
+-1*(L11-L21)+0*(L12-L22)+-1*(L13-L23)+1*(L14-L24)=0,
+0*(L11-L21)+-1*(L12-L22)+-1*(L13-L23)+0*(L14-L24)=0,
+-1*L21+0*L22+0*L23+0*L24=0,
+-1*L21+0*L22+-1*L23+0*L24=0,
+0*L21+-1*L22+0*L23+0*L24<0

Calculer les lambdas
Deuxième exemple
L2 * A1 = ((0.6))
L1 * b = ((-0.6))
L2 * b = ((-1))

Troisième exemple
L2 * A1 = ((0 1))
L1 * b = ((0))
L2 * b = ((-1))

Quatrieme exemple
L2 * A1 = ((0 1))
L1 * b = ((0))
L2 * b = ((-1))
```

A.3 Résoudre les systèmes d'inéquations

```
use_module(library(clpq)).

% Premier exemple :
% L1.2 :
{
    L11 >= 0, L12 >= 0, L13 >= 0, L14 >= 0,
    L21 >= 0, L22 >= 0, L23 >= 0, L24 >= 0,

    L12 - L13 - L14 = 0,
    -L11 - L21 + 2*L12 - 2*L22 - 2*L13 + 2*L23 + L14 -
        L24 = 0,
    -L21 + 3*L22 - 3*L23 = 0,
    10*L22 - 10*L23 < 0
}.
% L2.2 :
{
    L12 - L13 - L14 + 2*L15 - 2*L16 - L17 + 2*L18= 0,
    L15 - L16 - L18= 0,
    L25 - L26 - L28 = 0,
    -L11 + L21 + 2*L12 - 2*L22 - 2*L13 + 2*L23 + L17 -
        L27 = 0,
    -L21 + 3*L22 - 3*L23 - L24 + 2*L25 - 2*L26 + 2*L28
       = 0,
    10*L22 - 10*L23 + 10*L25 - 10*L26 < 0
}.

% Deuxième exemple :
% L :
{
    L11 >= 0, L12 >= 0, L13 >= 0,
    L21 >= 0, L22 >= 0, L23 >= 0,

    -2*L12 + 2*L13 = 0,
    -L11 + L21 - L12 + L22 + L13 - L23 = 0,
    -L21 - 3*L22 + 3*L23 = 0,
    -L21<0,

    L11 = 2 rdiv 3,
    L12 = 0,
    L13 = 0,

    L21 = 1,
```

```
    L22 = 0,
    L23 = 1 rdiv 3
}.

% Troisieme exemple :
% L :
{
    L11 >= 0, L12 >= 0, L13 >= 0, L14 >= 0, L15 >= 0,
       L16 >= 0,
    L21 >= 0, L22 >= 0, L23 >= 0, L24 >= 0, L25 >= 0,
       L26 >= 0,

    L12 - L13 - L16 = 0,
    L14 - L15 = 0,
    -L11 + L21 + L16 - L26 = 0,
    -L12 + L22 + L13 - L23 - L14 + L24 + L15 - L25 =
       0,
    -L21 + L22 - L23 = 0,
    -L22 + L23 = 0,
    -L24 + L25 < 0,

    L11 = 1,
    L12 = 1,
    L13 = 0,
    L14 = 0,
    L15 = 0,
    L16 = 1,

    L21 = 0,
    L22 = 0,
    L23 = 0,
    L24 = 1,
    L25 = 0,
    L26 = 0
}.

% Quatrieme exemple :
% L :
{
    L11 >= 0, L12 >= 0, L13 >= 0,
    L21 >= 0, L22 >= 0, L23 >= 0,

    L13 = 0,
    L12 = 0,
```

```
    -1*(L11-L21)-1*(L13-L23) = 0,
    -1*(L12-L22)-1*(L13-L23) = 0,
    -L21 = 0,
    -L23 = 0,
    -L22 < 0
}.

% Quatrieme exemple :
% L_synth :
{
    L11 >= 0, L12 >= 0, L13 >= 0, L14 >= 0,
    L21 >= 0, L22 >= 0, L23 >= 0, L24 >= 0,

    L13 - L14 = 0,
    -L11 + L12 = 0,
    -1*(L11-L21) - 1*(L13-L23) + 1*(L14-L24) = 0,
    -1*(L12-L22) - 1*(L13-L23) = 0,
    -L21 = 0,
    -L21 - L23 = 0,
    -L22 < 0,

    L11 = 0,
    L12 = 0,
    L13 = 1,
    L14 = 1,

    L21 = 0,
    L22 = 1,
    L23 = 0,
    L24 = 0
}.
```

A.4 Tests de satisfiabilité de l'inductivité d'un invariant de transition

```
from z3 import *

# Solveur pour le deuxieme exemple : R \subseteq T
x0, y0, x1, y1= Reals('x0 y0 x1 y1')
s2_1 = Solver()
s2_1.add(x0 > 1)
s2_1.add(-2*x1 == x0)
s2_1.add(Or(x0 < 0, x0 < (2/3)*x1))
print "Solveur pour le deuxieme exemple R \subseteq T"
print s2_1
print(s2_1.check())
print "\n"

# Solveur pour le deuxieme exemple : R \circ T \
   subseteq T
r0, t0, r1, t1 = Reals('r0 t0 r1 t1')
s2_2 = Solver()
s2_2.add(r0 > 1)
s2_2.add(-2*r1 == r0)
s2_2.add(t1 == r0)
s2_2.add(t0 >= 0)
s2_2.add(t0 >= (2/3)*t1)
s2_2.add(Not (t0 >=0, t0 >= 2*r1))
print "Solveur pour le deuxieme exemple R \circ T \
   subseteq T"
print s2_2
print(s2_2.check())
print "\n"

# Solveur pour le troisieme exemple : R \subseteq T
x0, y0, x1, y1= Reals('x0 y0 x1 y1')
s3_1 = Solver()
s3_1.add(x0 > 0)
s3_1.add(x1 == y0)
s3_1.add(y1 == y0 - 1)
s3_1.add(Not (x0 >= 0, x0 >= x1 + 1))
s3_1.add(Not (y0 >= 0, y0 >= y1 + 1))
print "Solveur pour le troisieme exemple R \subseteq T
   "
print s3_1
print(s3_1.check())
```

```
print "\n"

# Solveur pour le troisieme exemple : R \circ T \
    subseteq T
r0, t0, r1, t1, r02, t02, r12, t12 = Reals('r0 t0 r1
    t1 r02 t02 r12 t12')
s3_2a = Solver()
s3_2a.add(Or(And(t0 >= 0, And(t0 >= t1 + 1, And(t02 ==
     r0, And(t12 == r1, And(r0 > 0, And(r02 == r1, r12
    == r1 - 1)))))), (And(r02 >= 0, And(r02 >= r12 + 1,
     And(t02 == r0, And(t12 == r1, And(r0 > 0, And(r02
    == r1, r12 == r1 - 1)))))))))
s3_2a.add(Not(t0 >= 0, t0 >= t02 + 1))
s3_2a.add(Not (t1 >= 0, t1 >= t12 + 1))
print "Solveur pour le troisieme exemple R \circ T \
    subseteq T"
print s3_2a
print(s3_2a.check())
print "\n"

# Solveur pour le troisieme exemple : T \circ R \
    subseteq T
r0, t0, r1, t1, r02, t02, r12, t12 = Reals('r0 t0 r1
    t1 r02 t02 r12 t12')
s3_2b = Solver()
s3_2b.add(Or(And(r0 > 0, And(r02 == r1, And(r12 == r1
    - 1, And(t0 == r02, And( t1 == r12, And(t0 >= 0, t0
     >= t1 + 1)))))), (And(r0 > 0, And(r02 == r1, And(
    r12 == r1 - 1, And(t0 == r02, And(t1 == r12, And(
    t02 >= 0, t02 >= t12 + 1)))))))))
s3_2b.add(Not (r0 >= 0, r0 >= r1 + 1))
s3_2b.add(Not (t0 >= 0, t0 >= t1 + 1))
print "Solveur pour le troisieme exemple T \circ R \
    subseteq T"
print s3_2b
print(s3_2b.check())
print "\n"

# Solveur pour le quatrieme exemple : R \subseteq T
x0, y0, x1, y1= Reals('x0 y0 x1 y1')
s4_1 = Solver()
s4_1.add(x0 >= 0)
s4_1.add(y1 <= y0 - 1)
s4_1.add(x1 <= x0 + y0)
```

```
s4_1.add(Not (x0 >= 0, x0 >= x1 + 1))
s4_1.add(Not (y0 >= 0, y0 >= y1 + 1))
print "Solveur pour le quatrieme exemple R \subseteq T
    "
print s4_1
print(s4_1.check())
print "\n"

# Solveur pour le quatrieme exemple : R \circ T \
    subseteq T
r0, t0, r1, t1, r02, t02, r12, t12 = Reals('r0 t0 r1
    t1 r02 t02 r12 t12')
s4_2 = Solver()
s4_2.add(Or(And(t0 >= 0, And(t0 >= t02 + 1, And(t02 ==
     r0, And(t12 == r1, And(r0 > 0, And(r12 <= r1 - 1,
    r02 <= r0 + r1))))))), And(t1 >= 0, And(t1 >= t12 +
    1, And(t02 == r0, And(t12 == r1, And(r0 > 0, And(r1
     <= r1 - 1, r02 <= r0 + r1)))))))))
s4_2.add(Not (t0 >= 0, t0 >= t02 + 1) )
s4_2.add(Not (t1 >= 0, t1 >= t12 + 1))
print "Solveur pour le quatrieme exemple R \circ T \
    subseteq T"
print s4_2
print(s4_2.check())
```

A.5 Résultats des tests de satisfiabilité du code source de l'annexe A.4

```
Solveur pour le deuxieme exemple R \subseteq T
[x0 > 1, -2*x1 == x0, Or(x0 < 0, x0 < 0*x1)]
unsat


Solveur pour le deuxieme exemple R \circ T \subseteq T
[r0 > 1,
 -2*r1 == r0,
 t1 == r0,
 t0 >= 0,
 t0 >= 0*t1,
 Not(t0 >= 0)]
unsat


Solveur pour le troisieme exemple R \subseteq T
[x0 > 0, x1 == y0, y1 == y0 - 1, Not(x0 >= 0), Not(y0
   >= 0)]
unsat


Solveur pour le troisieme exemple R \circ T \subseteq
   T
[Or(And(t0 >= 0,
        And(t0 >= t1 + 1,
            And(t02 == r0,
                And(t12 == r1,
                    And(r0 > 0,
                        And(r02 == r1, r12 == r1 - 1))
                           )))),
    And(r02 >= 0,
        And(r02 >= r12 + 1,
            And(t02 == r0,
                And(t12 == r1,
                    And(r0 > 0,
                        And(r02 == r1, r12 == r1 - 1))
                           )))),
 Not(t0 >= 0),
 Not(t1 >= 0)]
sat
```

```
Solveur pour le troisieme exemple T \circ R \subseteq
   T
[Or(And(r0 > 0,
        And(r02 == r1,
            And(r12 == r1 - 1,
                And(t0 == r02,
                    And(t1 == r12,
                        And(t0 >= 0, t0 >= t1 + 1))))))
                           ),
    And(r0 > 0,
        And(r02 == r1,
            And(r12 == r1 - 1,
                And(t0 == r02,
                    And(t1 == r12,
                        And(t02 >= 0, t02 >= t12 + 1))
                           ))))),
 Not(r0 >= 0),
 Not(t0 >= 0)]
unsat


Solveur pour le quatrieme exemple R \subseteq T
[x0 >= 0,
 y1 <= y0 - 1,
 x1 <= x0 + y0,
 Not(x0 >= 0),
 Not(y0 >= 0)]
unsat


Solveur pour le quatrieme exemple R \circ T \subseteq
   T
[Or(And(t0 >= 0,
        And(t0 >= t02 + 1,
            And(t02 == r0,
                And(t12 == r1,
                    And(r0 > 0,
                        And(r12 <= r1 - 1, r02 <= r0 +
                            r1)))))),
    And(t1 >= 0,
        And(t1 >= t12 + 1,
            And(t02 == r0,
                And(t12 == r1,
```

```
                        And(r0 > 0,
                            And(r1 <= r1 - 1, r02 <= r0 +
                                r1))))))),
 Not(t0 >= 0),
 Not(t1 >= 0)]
unsat
```

B Lexique

Le lexique ci-dessous traduit certains mots de vocabulaire présent dans ce mémoire. Les termes anglais sont les termes utilisés dans les articles référencés dans la bibliographie et les termes français sont les termes utilisés dans ce mémoire. Ce sont les termes les plus spécifiques au domaine de la terminaison de boucles qui ont été traduits.

abstract interpretation algorithm	algorithme d'interprétation abstraite
binary reachability analysis	analyse d'accessibilité binaire
characterization	caractérisation
Chen's method	méthode de Chen
cutpoint	point de coupure
disjunctive invariance assertion	assertion invariante disjointe
disjunctive linear ranking function	fonction de rang linéaire disjonctive
disjunctive well foundedness principle	principe de disjonction bien fondée
disjunctively well-founded	disjointement bien fondée
eventual linear ranking function	fonction de rang linéaire à terme
finite difference (tree)	(arbre de) différence finie
invariance analyse	analyse invariante
invariance assertion	assertion invariante
lexicographic tuple of polynomials	tuple lexicographique polynomial
linear simple loop	boucle linéaire simple
local termination predicate	prédicat de terminaison locale
location	point de programme
loop termination	terminaison de boucles
piecewise-defined ranking function	fonction de rang définit par morceaux
polynomial assertion	assertion polynomiale
polynomial atom	atome polynomial
polynomial expression	expression polynomiale
polynomial loop	boucle polynomiale
polyranking function	fonctions à plusieurs rangs
ranking function	fonction de rang
rank function synthesis engines	synthétiseur de fonction de rang
ranking synthesizer	synthétiseur de rang
refinement	raffinement
smallest strongly-connected subgraph	plus petit sous-graphe fortement connecté
standard iterative fixpoint computations	générations itératives de point fixe
stem	tige
temporal safety checker	vérificateur de sécurité temporelle
trace	trace
Turing's proof	preuve de Turing
variance analyse	analyse variante
variance assertion	assertion variante